HAL
open science

Adaptation autonomique d'applications pervasives dirigée par les architectures

Etienne Gandrille

To cite this version:

Etienne Gandrille. Adaptation autonomique d'applications pervasives dirigée par les architectures. Informatique ubiquitaire. Université de Grenoble, 2014. Français. NNT : 2014GRENM078 . tel-01215004

HAL Id: tel-01215004

https://theses.hal.science/tel-01215004v1

Submitted on 13 Oct 2015

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.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Etienne GANDRILLE

Thèse dirigée par **Pr. Philippe LALANDA**
et codirigée par **Dr. Stéphanie CHOLLET**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique (MSTII)**

Adaptation autonomique d'applications pervasives dirigée par les architectures

Thèse soutenue publiquement le **12 décembre 2014**,
devant le jury composé de :

Pr. Ioannis PARISSIS
Professeur à Grenoble INP, Président
Pr. David R. C. HILL
Professeur à l'Université Blaise Pascal, Rapporteur
Pr. Lionel SEINTURIER
Professeur à l'Université de Lille I, Rapporteur
Dr. Mikael DESERTOT
Maître de conférences à l'Université de Valenciennes et du Hainaut-Cambrésis, Examinateur
Dr. Catherine HAMON
Ingénieur de recherche à Orange Labs, Invitée
Pr. Philippe LALANDA
Professeur à l'Université Joseph Fourier, Directeur de thèse
Dr. Stéphanie CHOLLET
Maître de conférences à Grenoble INP, Co-encadrante de thèse

Résumé

La problématique d'adaptation autonomique prend de plus en plus d'importance dans l'administration des applications modernes, notamment pervasives. En effet, la composition entre les différentes ressources de l'application (dispositifs physiques, services et applications tierces) doit souvent être dynamique, et s'adapter automatiquement et rapidement aux évolutions du contexte d'exécution. Pour cela, les composants orientés services offrent un support à l'adaptation au niveau architectural. Cependant, ils ne permettent pas d'exprimer l'ensemble des contraintes de conception qui doivent être garanties lors de l'exécution du système.

Pour lever cette limite, cette thèse a modélisé les architectures de conception, de déploiement et de l'exécution. De plus, elle a établi des liens entre celle-ci et proposé des algorithmes afin de vérifier la validité d'une architecture de l'exécution par rapport à son architecture de conception. Cela nous a conduits à considérer de près le cycle de vie des composants et à définir un ensemble de concepts afin de les faire participer à des architectures supportant la variabilité. Notons que cette formalisation peut être exploitée aussi bien par un administrateur humain, que par un gestionnaire autonomique qui voit ainsi sa base de connaissances augmentée et structurée.

L'implantation a donné lieu à la réalisation d'une base de connaissance, mise à disposition d'un atelier (Cilia IDE) de conception, déploiement et supervision d'applications dynamiques, ainsi que d'un gestionnaire autonomique capable de modifier la structure d'une application pervasive.

Cette thèse a été validée à l'aide d'une application pervasive nommée « Actimétrie », développée dans le cadre du projet FUI MEDICAL.

Mots-clés : adaptation, architecture, composant, informatique autonomique.

Abstract

The autonomic adaptation of software application is becoming increasingly important in many domains, including pervasive field. Indeed, the integration fo different application resources (physical devices, services and third party applications) often needs to be dynamic and should adapt rapidly and automatically to changes in the execution context. To that end, service-oriented components offer support for adaptation at the architectural level. However, they do not allow the formalisation of all the design constraints that must be guaranteed during the execution of the system.

To overcome this limitation, this thesis modeled the design, deployment and runtime architectures. Also, it proposes to establish links between them and has developed algorithms to check the validity of an execution architecture with respect to its architectural design. This led us to consider the entire life cycle of components and to define a set of concepts to be included in architectures supporting variability. This formalisation can be exploited both by a human administrator and by an autonomic manager that has its knowledge base increased and structured.

The implementation resulted in the realization of a knowledge base, providing a studio (Cilia IDE) for the design, deployment and supervision of dynamic applications, as well as an autonomic manager that can update the structure of pervasive applications.

This thesis has been validated using a pervasive application called "Actimetry", developed in the FUI MEDICAL project.

Keywords: adaptation, architecture, component, autonomic computing.

Sommaire

Chapitre 1 Introduction **1**

1.1 Informatique ubiquitaire 2

1.2 Auto-adaptation des applications ubiquitaires 5

1.3 Approche 7

1.4 Organisation du document 8

Partie I État de l'Art **9**

Chapitre 2 Adaptation logicielle **11**

2.1 La notion d'adaptation 13

2.2 Niveaux d'abstraction pour implanter l'adaptation 16

2.3 Services pour l'adaptation au niveau architectural 26

2.4 Lignes de produits logicielles 37

2.5 Lignes de produits dynamiques 46

2.6 Conclusion 56

Chapitre 3 Informatique autonomique **59**

3.1 Présentation de l'informatique autonomique 61

3.2 Sources d'inspiration 66

3.3 Les propriétés d'un système autonomique 70

3.4 Architecture des systèmes autonomiques selon IBM 73

3.5 Architectures pour représenter la connaissance 79

3.6 Conclusion 94

Partie II Contribution **95**

Chapitre 4 Proposition **97**
4.1 Problématique et objectifs 98
4.2 Approche 100
4.3 Etude de la variabilité dans les différentes architectures 106
4.4 Conclusion 109

Chapitre 5 Méta-modèles, modèles et algorithmes **111**
5.1 Définitions des éléments architecturaux 112
5.2 Formalisation des architectures 122
5.3 Formalisation des définitions et de leurs valeurs 129
5.4 Validation des architectures 138
5.5 Conclusion 154

Partie III Implantation et expérimentations **155**

Chapitre 6 Réalisation **157**
6.1 Introduction 158
6.2 La plate-forme Cilia 160
6.3 La base de connaissances 164
6.4 L'atelier Cilia IDE 166
6.5 Le gestionnaire autonomique de déploiement 170
6.6 Conclusion 172

Chapitre 7 Validation **173**
7.1 Introduction 174
7.2 L'application Actimétrie 177
7.3 Développement et supervision du service Actimétrie avec Cilia IDE . . . 184
7.4 Administration autonomique 193
7.5 Tests de performance 194
7.6 Conclusion 197

Partie IV Conclusion et Perspectives **199**

Chapitre 8 Conclusion et Perspectives **201**
8.1 Conclusion 202
8.2 Perspectives 204

Bibliographie **207**

Liste des figures

2.1 Adaptation et cycle de vie. 14
2.2 Exemple d'utilisation du préprocesseur. 16
2.3 Patron de conception stratégie. 18
2.4 Patron de conception pont. 19
2.5 Un système à composants [CSVC11]. 21
2.6 Substitution de composant [LDM13]. 22
2.7 Acteurs et interactions dans l'architecture à services [Pap03]. 27
2.8 SOC dynamique – arrivée d'un service. 29
2.9 SOC dynamique – départ d'un service. 30
2.10 Architecture orientée service : mécanismes d'implantation [EAA+04]. 31
2.11 Pyramide des fonctionnalités des architectures orientées service étendues (à partir de [PH07]). 32
2.12 Conteneur de Apache Felix iPOJO [EHL07]. 34
2.13 Vue simplifiée du conteneur de Apache Felix iPOJO. 35
2.14 Diagramme de fonctionnalités [KCH+90]. 39
2.15 Coûts de développement avec ou sans ligne de produits [PBvdL05]. 44

3.1 Autonomic computing adoption model [IBM06]. 64
3.2 Boucle de rétroaction. 67
3.3 Arbre de l'informatique autonomique [SB03a]. 71
3.4 Elément autonomique. 74
3.5 Boucle de contrôle MAPE-K. 76

3.6 Le *framework* Rainbow [GSC09]. 80
3.7 Architecture de Znn.com [GSC09]. 83
3.8 Vue de haut niveau de l'approche Plastik [GBJC07]. 85
3.9 Architecture à trois couches de J. Kramer et J. Magee [KM09]. 87
3.10 Architecture à trois couches, revisitée par [MMMR12]. 89
3.11 Vue de haut niveau de l'approche ARCM [GvdHT09]. 91

4.1 Principes de notre approche. 101
4.2 Conformité entre les modèles et les méta-modèles. 101
4.3 Organisation générale des méta-modèles. 102
4.4 Niveaux de variabilité dans les différentes architectures. 102
4.5 Les transformations entre méta-modèles. 103
4.6 Vérification de la validité de l'exécution par rapport à la conception. 104
4.7 Illustration de la variabilité topologique. 106
4.8 Illustration de la variabilité au niveau d'un composant. 107
4.9 Illustration de la variabilité au niveau de la configuration d'un composant. . . 107

5.1 Exemple introductif aux types de composant et aux composants. 112
5.2 Illustration des interfaces et des contrats entre composants. 113
5.3 Les composants utilisés dans les différentes architectures. 114
5.4 Utilisation des propriétés pour les types de composants. 115
5.5 Utilisation des propriétés avec les composants et leurs types. 116
5.6 Utilisation des paramètres. 117
5.7 Utilisation des variables d'état. 118
5.8 Exemple d'utilisation de contrainte sur une propriété. 119
5.9 Exemple d'utilisation de contrainte sur un paramètre. 119
5.10 Exemple d'utilisation de contrainte sur une variable d'état. 120
5.11 Méta-modèle commun définissant une architecture. 122
5.12 Méta-modèle de l'architecture de conception. 125
5.13 Méta-modèle de l'architecture de déploiement. 126
5.14 Méta-modèle de l'architecture de l'exécution. 127

5.15 Ensemble des variations possibles autour des classes *Définition* et *Valeur* du méta-modèle commun. 129
5.16 Diagramme d'objets exprimant un ensemble de propriétés. 131
5.17 Diagramme d'objets exprimant un ensemble de paramètres. 132
5.18 Diagramme d'objets exprimant un ensemble de variables d'état. 135
5.19 Diagramme d'objets exprimant une contrainte. 137
5.20 Les transformations de modèles. 138
5.21 Illustration du résultat souhaité par l'algorithme de correspondance. 139
5.22 Illustration du parcours des liaisons pour le calcul des correspondances. . . . 143
5.23 Exemple pour l'explication de l'algorithme verification-cardinalité-liaison. . . 151

6.1 Modules de notre réalisation. 159
6.2 Exemple de chaîne de médiation Cilia. 160
6.3 Architecture d'un médiateur Cilia. 160
6.4 Architecture des adaptateurs Cilia. 161
6.5 Architecture de *Cilia Mediation Framework*. 162
6.6 Architecture réflexive de *Cilia Mediation Framework* (modèle simplifié). 162
6.7 L'atelier Cilia IDE. 166
6.8 Présentation de la vue *Implementations* de Cilia IDE. 167
6.9 Présentation d'une vue architecture de Cilia IDE. 168
6.10 Propriétés de l'implantation *MeasureFilterMediator*. 169
6.11 Vue *Cilia error view*. 169

7.1 Un défi du *Smart Home*. 174
7.2 Architecture de la plate-forme iCASA. 176
7.3 Les deux parties de l'application Actimétrie. 178
7.4 Capture d'écran du tableau de bord de l'application d'actimétrie. 179
7.5 Déploiement du service Actimétrie dans plusieurs habitats. 179
7.6 Médiation de l'Actimétrie déployée dans la maison d'Aurélie. 181
7.7 Etapes du déploiement du service Actimétrie. 182
7.8 Représentation de l'architecture de conception du service Actimétrie. 184
7.9 Représentation de l'architecture de déploiement du service Actimétrie. 186

7.10 Définition de l'adresse de la plate-forme distante dans Cilia IDE. 188
7.11 Choix du fichier dscilia à transférer. 189
7.12 Visualisation de la chaîne déployée avec Cilia IDE. 189
7.13 Informations générales du médiateur de filtrage. 189
7.14 Propriétés du médiateur de filtrage. 190
7.15 Variables d'états du médiateur de filtrage. 190
7.16 Mise en relation de l'architecture d'exécution avec l'architecture de conception. 191
7.17 Détection d'erreur de validation dans Cilia IDE. 191
7.18 Visualisation du lien entre les architectures dans Cilia IDE. 192
7.19 Les deux gestionnaires autonomiques de notre validation. 193
7.20 Exemple de réification de la chaîne à l'exécution avant et après intervention du gestionnaire autonomique de déploiement. 194
7.21 Architecture de type chaîne. 195
7.22 Temps de calcul des correspondances dans une architecture de type chaîne. . 195
7.23 Architecture sans liaison. 196
7.24 Temps de calcul des correspondances dans une architecture sans liaison. . . . 196

Liste des tableaux

2.1 Résumé de l'approche SESAMO 49
2.2 Résumé de l'approche MADAM 50
2.3 Résumé de l'approche de Trinidad *et al.* 50
2.4 Résumé de l'approche Genie. 51
2.5 Résumé de l'approche CAPucine. 52
2.6 Résumé de l'approche DiVA 53
2.7 Résumé de l'approche MUSIC 54
2.8 Résumé de l'approche SASSY 55
2.9 Résumé de l'approche Lee et al 55

3.1 Résumé de l'approche Rainbow. 84
3.2 Résumé de l'approche Plastik. 86
3.3 Résumé de l'approche de J. Kramer et J. Magee. 88
3.4 Résumé de l'approche Prism-MW et DIF. 90
3.5 Résumé de l'approche ARCM. 92

4.1 Comparaison des différentes architectures. 105

5.1 Comparaison des composants et de leurs types. 121

6.1 Nombre de classes et de lignes de code de notre réalisation 172

7.1 Temps de reconfiguration de la plate-forme Cilia par opération [GGMD+11]. 197

Chapitre 1

Introduction

Sommaire

1.1 Informatique ubiquitaire 2
1.2 Auto-adaptation des applications ubiquitaires 5
1.3 Approche 7
1.4 Organisation du document 8

1.1 Informatique ubiquitaire

L'informatique ubiquitaire a été introduite par M. Weiser en 1991 [Wei91]. Elle propose un nouveau mode d'interaction entre l'homme et la machine basé sur un grand nombre de capteurs et d'actionneurs, disséminés dans l'environnement, et utilisables de manière simple et transparente. C'est donc une évolution de l'informatique telle qu'on la conçoit habituellement, qui utilise un ensemble limité d'écrans dédiés (ordinateurs, téléphones, tablettes...). Plus précisément, dans [WB97], M. Weiser place l'informatique ubiquitaire comme la dernière étape de l'évolution de l'informatique :

- à l'époque du *mainframe*, plusieurs personnes devaient partager un même ordinateur ;
- à l'époque de l'ordinateur individuel, chacun pouvait disposer de son ou ses ordinateurs qu'il possédait personnellement ;
- à l'époque de l'informatique ubiquitaire, nous aurons tous un large nombre d'ordinateurs que nous porterons sur nous ou qui seront intégrés à notre environnement quotidien.

Selon M. Weiser, en disséminant de manière naturelle l'information dans notre environnement, notre cerveau sera moins sollicité, tout en conservant la disponibilité immédiate de l'information. Ce concept, qu'il définit de calme technologique, est selon lui l'avantage central de l'ubiquitaire :

> "The most potentially interesting, challenging, and profound change implied by the ubiquitous computing era is a focus on calm. If computers are everywhere they better stay out of the way, and that means designing them so that the people being shared by the computers remain serene and in control. Calmness is a new challenge that UC [ubiquitous computing] brings to computing."

Dans le sillage de M. Weiser, un très large ensemble de travaux se sont développés et ont conduit à l'apparition de nombreux termes tels que informatique pervasive, intelligence ambiante, internet des objets, choses qui pensent... D'après D. Ronzani [Ron09] qui a étudié l'utilisation de ceux-ci dans les médias de masse, ils reprennent tous l'idée centrale d'un grand nombre de dispositifs communicants répartis dans l'environnement, tout en mettant l'accent sur des points différents : l'informatique ubiquitaire est souvent associée aux environnements professionnels, l'informatique pervasive aux réseaux, l'intelligence ambiante aux capteurs intelligents. De plus, D. Ronzani constate que le sens même de certains de ces mots a évolué au cours du temps.

Nous allons à présent explorer quelques définitions, afin de cerner plus précisément les contours de l'informatique ubiquitaire :

- **la transparence pour l'utilisateur :** si M. Weiser théorise la disparition de la technologie dans l'environnement, la plupart des auteurs parlent d'interactions plus fluides, plus naturelles qui permettent à l'utilisateur de se centrer sur la tâche à accomplir, et non la manière de l'accomplir :

> "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it. (...) Prototype tabs, pads and boards are just the beginning of ubiquitous computing. The real power of the concept comes not from any one of these devices—it emerges from the interaction of all of them." [Wei91]
>
> "Pervasive, or ubiquitous, computing has the potential to radically transform the way people interact with computers. (...) Pervasive computing thus marks a major shift in focus, away from the actual computing technology and towards people and their needs. So, instead of manually managing their computing environment by, for example, copying files between devices or converting between data formats, users "simply" access their applications and their data whenever and wherever they need." [GDL+04]

- **la collaboration :** la puissance du système global réside dans la capacité de collaborations des parties qui le composent, qui communiquent directement entre elles, afin de partager de l'information et exécuter des interactions plus ou moins complexes. Cela est mis en avant dans [Wei91], cité précédemment.
- **la distribution, la connexion et l'omniprésence :** l'ensemble des composants d'un système ubiquitaire est distribué et connecté à travers un ou plusieurs réseaux (fixes ou mobiles) qui couvrent si possible la totalité de l'environnement d'un usager, à chaque instant :

> "The essence of that vision [M. Weiser vision] was the creation of environments saturated with computing and communication capability, yet gracefully integrated with human users." [Sat01]
>
> "From a technological point of view, one could describe "ubiquitous computing" as the prospect of connecting the remaining things in the world to the Internet, in order to provide information "on anything, anytime, anywhere." Putting it in another way, the term "ubiquitous computing" signifies the omnipresence of tiny, wirelessly interconnected computers that are embedded almost invisibly into just about any kind of everyday object." [Mat01]

- **la sensibilité au contexte :** le système doit être capable de prendre en compte les évolutions du contexte d'exécution, telle que la disponibilité des équipements, les variations de mesures physiques (environnement ou paramètre physiologiques de l'utilisateur) :

> "First, [ubiquitous] systems need to embrace contextual change, so that applications can implement their own strategies for handling changes. Second, systems need to encourage ad hoc composition, so that applications can be dynamically connected and extended in an ever changing runtime environment. Third, systems need to recognize sharing as the default, so that applications can make information accessible anywhere and anytime." [GDL+04]

Du point de vue des capacités du matériel, la situation a considérablement évolué depuis le constat initial de M. Weiser et les défis associés à leur fabrication sont déjà largement levés :

- **la miniaturisation :** nous disposons aujourd'hui d'équipements de très faible taille comme par exemple le microcontrôleur STM32W, qui contient dans un carré de six millimètres de côté un processeur 32 bits, 256 Mo de mémoire vive, un module de cryptage et un circuit pour la communication sans fil ZigBee.
- **la communication :** de nombreux protocoles de communication sans fil ont vu le jour (Wi-Fi [Dor02], ZigBee [All12], 6LoWPAN [SB11], EnOcean [PRK10], Z-Wave [JFR06]) pour connecter des systèmes sur une plage de distance de quelques centimètres à plusieurs centaines de mètres, et même plus à l'aide des réseaux de téléphonie mobile 3G [1] et 4G [2].
- **l'autonomie :** si la consommation énergétique reste un élément crucial, il existe aujourd'hui des capteurs capables de transmettre la pression ou la température sur une portée de quelques mètres, sans source énergétique interne. L'énergie nécessaire à la transmission des données, et aussi, pour ajouter les données d'identification est obtenue à travers le processus de mesure lui-même, en utilisant des matériaux piézoélectriques ou pyroélectriques [Mat01].
- **le coût :** la diminution du coût du matériel ne cesse de se poursuivre, à mesure que la densité en terme de transistors augmente selon la loi de Moore.

1. https://en.wikipedia.org/wiki/3G
2. https://en.wikipedia.org/wiki/4G

1.2 Auto-adaptation des applications ubiquitaires

Comme nous l'avons vu, les applications ubiquitaires utilisent un grand nombre de capteurs et d'actionneurs disséminés dans l'environnement. La composition entre les différentes ressources de l'application (dispositifs physiques, services et applications tierces) doit être dynamique et s'adapter automatiquement et rapidement aux changements du contexte d'exécution. Ainsi, les applications ubiquitaires peuvent présenter des architectures différentes au cours du temps afin de rendre le même service.

Cette adaptation à l'exécution est largement identifiée comme d'une grande complexité [RCAM+05, SHB10]. En même temps, c'est le défi clé à relever pour bâtir des applications pleinement ubiquitaires [GDL+04]. De façon pratique, les adaptations architecturales en cours d'exécution demandent de :

- suivre et d'interpréter le contexte d'exécution ;
- décider des adaptations à apporter ;
- vérifier la réussite des adaptations ;
- éventuellement, déployer et installer du code afin d'intégrer de nouveaux éléments ou de nouvelles données.

Les travaux qui traitent de l'adaptation à l'exécution des applications ubiquitaires s'inscrivent dans le contexte plus large de l'étude des systèmes auto-adaptables. Un système auto-adaptable peut être défini comme un système qui se modifie lui-même à l'exécution pour assurer la satisfaction d'objectifs dans un environnement mouvant [VMT11].

Dans ce cadre, un grand nombre de travaux s'est intéressé à l'utilisation d'architectures [GCH+04, FHS+06, BGF+08, FDB+12]. En effet, les architectures sont apparues très tôt [OGT+99] comme un moyen approprié afin de détecter les modifications à apporter, les planifier et les exécuter.

Les approches actuelles privilégient la notion de composant logiciel et les approches à service. Les composants, apparus dans les années 1990, ont pour vocation de servir comme éléments de base pour le développement d'applications logicielles [Szy97]. Plus précisément, les approches à composants perçoivent le développement d'applications logicielles comme un assemblage de composants, et gèrent la maintenance et l'évolution d'applications par la personnalisation et le remplacement de composants réutilisables. Un composant est souvent proposé sous la forme d'une unité de déploiement qui encapsule un certain nombre d'éléments parmi lesquels son implantation. Un composant possède un ensemble d'interfaces. Ces interfaces définissent les fonctionnalités fournies et requises d'un composant sous la forme d'un contrat spécifique et visible de l'extérieur. Des modèles spécifiques à composants définissent la structure standard des composants, la manière de réaliser des assemblages et fournissent les mécanismes nécessaires pour la prise en compte de certains aspects non-fonctionnels tels que la distribution, la sécurité ou bien encore les transactions. Les applications basées sur les composants emploient les contrats des composants pour décrire leurs interactions et leurs dépendances fonctionnelles sous un contexte donné avec une

vision structurale. Cette représentation explicite de l'architecture élève le niveau d'abstraction par rapport aux assemblages d'objets, puisque la granularité de développement devient plus grande lors du passage de l'objet au composant. Par ce niveau élevé de granularité, les composants favorisent les adaptations à l'exécution. Cependant, les assemblages restent guidés par l'administrateur et ne peuvent être réalisés par les composants eux mêmes.

Les composants orientés service [Cer04] apportent une solution à cette limite. Les approches orientées services ont été introduites comme un nouveau paradigme pour le développement logiciel durant les années 2000. Ce paradigme utilise la notion de service comme élément de base pour la construction d'applications logicielles. Un service est défini comme une entité logicielle qui peut être utilisée de façon statique ou dynamique pour la réalisation d'une application logicielle. Un consommateur, ou client, sélectionne un service à partir de sa description. Il l'utilise sans avoir connaissance de la technologie sous-jacente nécessaire à son implantation ni de sa plate-forme d'exécution. Inversement, le service ne connaît pas le contexte dans lequel il va être utilisé par un client. Cette indépendance à double sens est une propriété forte des services qui facilite le faible couplage.

L'approche à composants orientés service proposée par [CH04] introduit l'approche orientée service dans l'approche orientée composant. Un composant orienté service est un composant conteneur d'aspects fonctionnels et non-fonctionnels dont les connecteurs suivent le style architectural de l'approche orientée service. Le code fonctionnel est exécuté à l'intérieur d'un conteneur qui va prendre en charge les propriétés non-fonctionnelles. Cette approche permet la simplification de l'écriture des applications orientées service. Le développeur n'a pas à prendre en compte la gestion de la disponibilité dynamique ainsi que le cycle de vie du composant. Ces tâches sont déléguées au code non-fonctionnel. Un modèle à composants orientés service ajoute au composant conteneur des interfaces de contrôle, des propriétés de service ainsi que des dépendances de déploiement. Les interfaces de contrôle permettent la gestion de la reconfiguration dynamique, les propriétés de services sont associées à l'implantation du composant. Les dépendances de déploiement assurent les dépendances de ressources gérées par l'environnement d'exécution.

Avec les composants orientés service, une partie des adaptations est directement décidée et prise en charge par la plate-forme d'exécution. Ceci est très intéressant en termes de réactivité et de modularité. L'administrateur ou le module en charge de l'administration n'est pas obligé de régulièrement mettre à jour l'application lui-même. Ceci pose néanmoins un réel problème d'administration, à savoir le suivi et la compréhension de l'architecture d'exécution. En effet, puisque les décisions d'évolution sont prises par la plate-forme, l'administrateur peut perdre le fil et ne plus connaître ou comprendre l'architecture à l'exécution. Cette thèse traite plus particulièrement de ce dernier point.

1.3 Approche

Nous avons vu que les composants apportaient une structuration claire du code, mais ne prenaient pas en charge la dynamicité à l'exécution. Pour cela, les composants orientés services sont apparus dans un second temps. Avec cette approche, il est possible de faire évoluer une application lors de son exécution, en établissant dynamiquement des liaisons entre les composants grâce à l'approche à services.

L'évolution de l'application n'est alors guidée que par les règles syntaxiques de correspondance d'interfaces de service : toute liaison valide, dans laquelle le contrat de service proposé par un composant correspond à une dépendance d'un autre composant peut être établie. Cela est clairement insuffisant et il nous apparaît nécessaire d'ajouter un jeu de contraintes supplémentaires pour que l'architecture de l'application demeure dans un périmètre plus restreint, défini lors de la conception en fonction du but poursuivi par l'application. Récemment, quelques travaux ont exploré cette voie, en particulier dans le cadre des lignes de produits logicielles dynamiques [MBJ+09]. Dans cette approche inspirée des lignes de produits logicielles [CN02], les points de variabilité sont résolus à l'exécution, là où ils l'étaient à la conception pour les lignes de produits classiques.

Toutefois, cette approche repose souvent sur un ensemble de sous-systèmes statiquement définis, qu'il est possible d'instancier selon un jeu de contraintes définis dans un modèle du domaine. On perd donc une partie du degré de flexibilité introduit par l'approche à service.

Pour retrouver pleinement celui-ci, nous nous sommes placés dans l'héritage de R. France et B. Rumple [FR07], qui défendent les approches à base de modèles en relation. Plus précisément, nous souhaitons lier une architecture de l'exécution à une architecture de conception, tout en restant au niveau d'abstraction des modèles à composants. L'architecture de conception pourrait ainsi présenter de la variabilité, qui serait résolue à l'exécution. Cette variabilité pourrait s'exprimer au niveau de la structure de l'application, en définissant des parties obligatoires, optionnelles ou multiples (comme dans les lignes de produits). Elle pourrait aussi s'exprimer selon les concepts de l'approche à services en définissant lors de la conception des spécifications de composants qui appelleront une sélection d'implantations à l'exécution.

Pour cela, il est donc nécessaire de définir clairement la notion de composant au long de son cycle de vie. Qu'est-ce qu'un composant spécifié ? implanté ? exécuté ? Quels sont les liens qui unissent ceux-ci ? Il est également nécessaire de construire et de maintenir les liens entre spécifications et exécutions de façon à guider efficacement l'administrateur.

De par l'accent mis sur l'exécution, les travaux autour des composants ne se sont pas penchés de manière approfondie sur ces questions. Or, celles-ci nous semblent conditionner la possibilité de tirer harmonieusement parti de l'approche à composants dans des systèmes dynamiques qui utilisent une architecture de conception pour guider et valider l'architecture d'exécution.

Cette thèse propose donc d'explorer cette voie, en validant cette approche sur un cas d'usage issu du projet FUI MEDICAL (`http://medical.imag.fr`).

1.4 Organisation du document

Le document est divisé en trois grandes parties : l'état de l'art, la contribution et, finalement, l'implantation et les expérimentations.

L'état de l'art est présenté en deux chapitres :

- le **chapitre 2 page 11** traite de la notion d'adaptabilité en logiciel. Il définit cette notion, montre les différents moyens de mettre en place des adaptations et présente les approches aujourd'hui disponibles dans le domaine de l'informatique ubiquitaire.
- le **chapitre 3 page 59** se concentre sur la notion d'informatique autonomique. Il définit précisément cette approche et parcourt ses sources d'inspiration. Il décrit également les systèmes existants et insiste sur la notion de connaissance qui est à la base de cette approche d'auto-administration.

La contribution est présentée en deux chapitres :

- le **chapitre 4 page 97** présente notre proposition qui s'appuie sur un ensemble d'architectures en relation ;
- le **chapitre 5 page 111** détaille les méta-modèles, modèles et algorithmes introduits au chapitre précédent.

La partie implémentation et expérimentations est présentée en deux chapitres :

- le **chapitre 6 page 157** présente les réalisations logicielles liées à cette thèse. Nous présentons en particulier un atelier permettant de faire graphiquement le lien entre architectures de conception et d'exécution.
- le **chapitre 7 page 173**, enfin, traite de la validation de notre approche. Il présente le projet FUI MEDICAL et décrit le cas d'utilisation principal. Il montre ensuite les différentes architectures liées à ce cas d'utilisation et leurs liens calculés.

Enfin, le **chapitre 8 page 201** fait le point sur nos principales contributions et indique quelques perspectives possibles de nos travaux.

Première partie

État de l'Art

Chapitre 2

Adaptation logicielle

Sommaire

2.1 La notion d'adaptation **13**
2.1.1 Introduction 13
2.1.2 Adaptation statique et adaptation dynamique 14
2.1.3 Niveaux d'abstraction pour implanter l'adaptation 15
2.2 Niveaux d'abstraction pour implanter l'adaptation **16**
2.2.1 Adaptation au niveau code 16
2.2.2 Adaptation au niveau objet 17
2.2.3 Adaptation au niveau composant 20
2.2.4 Adaptation au niveau architecture logicielle 22
2.2.5 Synthèse 25
2.3 Services pour l'adaptation au niveau architectural **26**
2.3.1 Notion de service 26
2.3.2 Approche orientée service 27
2.3.3 Architecture orientée service 31
2.3.4 Approche à composants à services 33
2.3.5 Synthèse 36
2.4 Lignes de produits logicielles **37**
2.4.1 Introduction 37
2.4.2 L'ingénierie du domaine 37
2.4.3 Ingénierie d'applications 42
2.4.4 Bénéfices liés à l'utilisation de lignes de produits 43
2.4.5 Synthèse 45
2.5 Lignes de produits dynamiques **46**
2.5.1 Introduction 46
2.5.2 Critères de caractérisation 47

2.5.3 Différents travaux 48
2.5.4 Synthèse des lignes de produits dynamiques 56
2.6 Conclusion 56

Le but de ce chapitre est de présenter la notion d'adaptation logicielle. Pour cela, nous allons commencer par présenter la définition de ce concept, ainsi que les différents niveaux d'abstraction auxquels il est possible de traiter l'adaptation : le code source, les objets, les composants puis les architectures. Nous verrons que le meilleur niveau pour traiter cette préoccupation est le niveau architectural ; nous présenterons donc l'approche à services qui s'inscrit dans cette perspective. Cependant, un manque de contrôle dans la sélection des services nous fera nous intéresser aux lignes de produits statiques, puis dynamiques, qui prennent particulièrement bien en compte cette préoccupation, lors de la conception pour les premières et jusqu'à l'exécution pour les secondes.

2.1 La notion d'adaptation

2.1.1 Introduction

Il est rare qu'un logiciel soit écrit une fois pour toutes et qu'aucune modification ne lui soit apportée après son déploiement et sa mise en opération. Il est au contraire souvent nécessaire de modifier sa structure ou son comportement alors que celui-ci est déjà exécuté. Le but de l'adaptation est de prendre en charge ces modifications, afin de permettre à un logiciel en exécution de fonctionner en adéquation avec de nouvelles ressources ou de nouveaux besoins.

> "Software adaptation is a discrete process allowing a software system to continuously meets its goals in a changing environment." [LDM13]

L'adaptation trouve donc son origine dans l'évolution (et souvent, l'élargissement) des besoins sur le plan fonctionnel ou non-fonctionnel. De plus, elle peut aussi viser la prise en charge de la correction d'erreurs détectées lors de l'utilisation du logiciel.

Il est important de noter que cette demande d'adaptation est de plus en plus importante [Gar13] : l'adaptation permanente des entreprises à un contexte mouvant dans lequel les ressources évoluent fréquemment passe par l'adaptation en continu des logiciels employés. Dans ce contexte, les méthodes agiles ont cherché depuis plusieurs années à prendre en charge au plus tôt ces modifications de besoins exprimés, en livrant très régulièrement un produit fonctionnel. Plus récemment, le déploiement continu [HF10] a proposé de déployer jusqu'à pluri quotidiennement un logiciel en cours de développement, en outillant et en automatisant l'ensemble du processus de compilation, de déploiement et de configuration. Cependant, cette approche extrêmement intégrée ne traite pas l'adaptation lors de l'exécution.

De manière plus générale, bien que la capacité d'adaptation soit de plus en plus demandée, son implantation repose bien souvent sur des mécanismes ad hoc. De plus, l'implantation du mécanisme d'adaptation est souvent non préparée car difficile à prévoir. Elle nécessite généralement des modifications de code qui peuvent être importantes et conduire

à des effets de bords non prévus. De plus, ces modifications peuvent avoir des conséquences en de multiples points du code de l'application, même pour une modification qui pouvait sembler localisée. Cette adaptation est donc d'autant plus complexe et demande une expertise et des efforts importants.

2.1.2 Adaptation statique et adaptation dynamique

L'adaptation d'un logiciel peut être effectuée à chaque étape de son cycle de vie (figure 2.1). Lors de l'écriture du code, mais aussi lors de la phase de compilation (compilation conditionnelle), d'édition des liens, de *packaging*, de déploiement (sélection des artéfacts à lier, *packager*, déployer), de configuration, de lancement de l'application (paramétrage), ou lorsque celle-ci est déjà en exécution. A chaque étape se rapporte un ensemble de techniques que l'on peut retrouver dans des classifications telles que [SvGB05, GH12, CKB13].

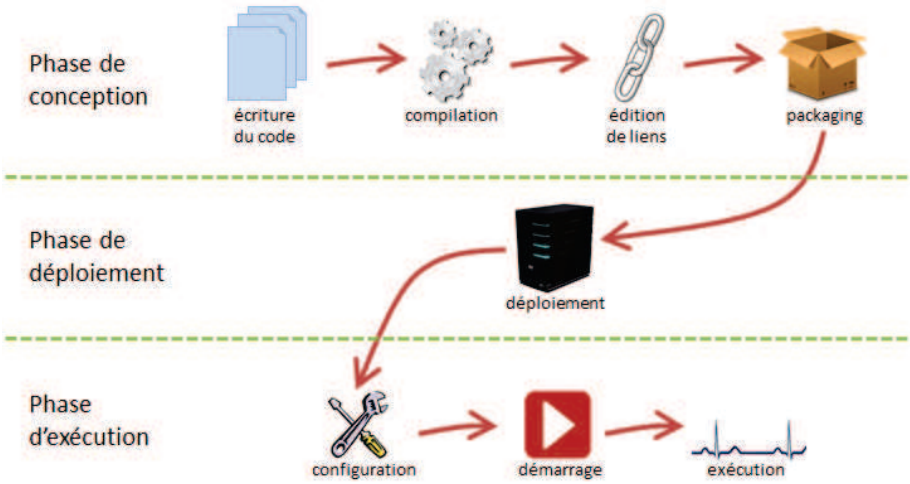

FIGURE 2.1 – Adaptation et cycle de vie.

L'adaptation peut donc être effectuée avec ou sans redémarrage du système. On parle ainsi d'**adaptation statique**, quand le système est redémarré dans le cadre de cette opération. L'adaptation a alors lieu lors des phases de développement, de compilation ou de lancement de l'application. De manière symétrique, **on qualifie l'adaptation de dynamique** quand ce processus a lieu sans arrêt du système. Cela signifie que des éléments de l'application comme des structures ou des algorithmes peuvent être ajoutés, supprimés ou remplacés « à chaud ». Cela implique l'utilisation de plates-formes d'exécution qui permettent de prendre en charge ces opérations. Par exemple, certaines plates-formes à composants dynamiques permettent d'ajouter ou de supprimer des composants à l'exécution et de modifier dynamiquement les liaisons entre ceux-ci [EHL07].

L'adaptation dynamique est plus complexe que l'adaptation statique à mettre en œuvre. En effet, celle-ci nécessite de préserver les flots de données et de contrôle, ainsi que les états, alors que l'application peut être modifiée en profondeur. Par exemple, dans le cadre d'un site Internet marchand, une adaptation de l'application ne doit pas faire perdre le contenu des

paniers des clients connectés sur le site. Si l'adaptation statique est plus simple à mettre en œuvre, elle fait cependant baisser la disponibilité des systèmes. Aussi, cette approche doit être écartée au profit de l'adaptation dynamique quand cette disponibilité doit être totale, ou qu'une série d'indisponibilités (même courtes) n'est pas possible ou acceptable dans le contexte d'utilisation du service.

2.1.3 Niveaux d'abstraction pour implanter l'adaptation

Que l'on soit dans le cadre de l'adaptation statique ou dynamique, l'implantation de l'adaptation peut être effectuée à plusieurs niveaux d'abstraction :

- **au niveau code**, avec une granularité extrêmement fine : la ligne de langage de programmation ;
- **au niveau objet**, avec une granularité plus importante où le code est regroupé dans un concept, souvent de bas niveau ;
- **au niveau composant**, à l'aide d'entités de plus grande taille permettant de manipuler en un grain unique une quantité importante de code, tout en exprimant de manière explicite des dépendances (requises et fournies) ;
- **au niveau architecture**, qui offre une perspective globale sur l'ensemble de l'application.

Nous allons à présent détailler ces niveaux d'abstraction en présentant pour chacun des mécanismes d'adaptation. Cela mettra en lumière celui qui est le plus adapté afin d'implanter le mécanisme d'adaptation.

2.2 Niveaux d'abstraction pour implanter l'adaptation

2.2.1 Adaptation au niveau code

L'adaptation au niveau du code vise à modifier de manière extrêmement ciblée les lignes ou blocs de code qui seront exécutés. C'est donc le niveau d'abstraction le plus bas qui est aujourd'hui employé. Ce niveau d'adaptation permet des modifications extrêmement fines et ciblées, mais demande en contrepartie la connaissance du langage de programmation, et donc une importante technicité. Une liste complète de ces techniques peut être trouvée dans [eAMO11]. Nous allons nous focaliser sur deux d'entre elles, qui permettront de comprendre le type de mécanismes en jeu.

La première utilise des directives de compilation conditionnelles. Celles-ci permettent de guider la phase de compilation en fonction de certaines conditions. Par exemple, dans le langage C, elles tirent parti du préprocesseur qui va inclure ou supprimer d'un fichier certaines lignes. Cela est souvent utilisé afin de permettre l'inclusion conditionnelle de ressources, notamment afin de compiler un programme pour un système d'exploitation particulier, en incluant les bonnes bibliothèques graphiques ou d'accès au matériel.

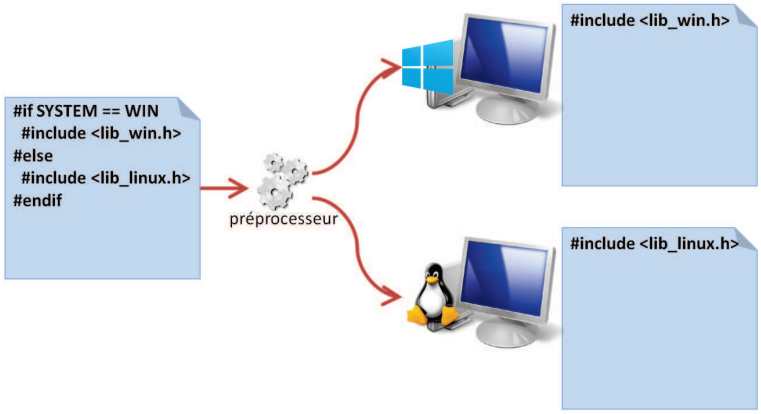

FIGURE 2.2 – Exemple d'utilisation du préprocesseur.

Dans l'exemple de la figure 2.2, nous pouvons constater que le code qui sera compilé après passage du préprocesseur est adapté à la plate-forme d'exécution cible. Les différents variants sont exprimés dans le code et sélectionnés au moment du passage du préprocesseur. Après celui-ci, il n'est plus possible de revenir sur les choix effectués et la liaison est donc permanente.

La seconde technique que nous allons présenter utilise les pointeurs de fonctions. Ceux-ci permettent de choisir jusqu'au moment de l'appel l'implantation d'une fonction donnée par modification de pointeur. Cette technique peut être mise en œuvre dans de nombreux langages, et notamment le langage C où elle a été beaucoup employée comme dans [NHSO06].

```
1 int main(int argc, char * argv[]) {
2   // initalisation du tableau et de son nombre d'elements
3   qsort(tab, nbElem, sizeof(int), compFunction);
4   // suite du programme
5 }
6
7 int compFunction(const void *a, const void *b) {
8   return abs(*((int*)b)) - abs(*((int*)a));
9 }
```

Code source 2.1 – Utilisation de pointeur de fonction.

Le Code source 2.1 montre comment on peut adapter la fonction *qsort* en lui donnant un pointeur vers une fonction de tri. Avec cet exemple, les nombres d'un tableau d'entiers seront triés par ordre croissant des valeurs absolues.

En conclusion, les mécanismes d'adaptation au niveau code offrent une très grande finesse dans le processus d'adaptation, en permettant de décider précisément quelles lignes de code seront exécutées. Cependant, la mise en œuvre de l'adaptation repose ici directement sur les capacités du langage. Pour simplifier l'écriture et fiabiliser le code produit, il apparaît souhaitable de disposer de modèles éprouvés pour guider l'implantation des mécanismes d'adaptation. C'est là l'objectif des patrons de conception de l'approche orientée objet, que nous allons maintenant présenter.

2.2.2 Adaptation au niveau objet

2.2.2.1 Introduction

L'origine de l'approche orientée objet remonte aux années 1960, avec le langage SIMULA [Hay03] qui sera suivi peu de temps après par la création du langage Smalltalk [GR83]. Au centre de cette approche se trouve la notion d'objet :

> « Un objet est une abstraction d'un concept existant dans la réalité. Concrètement, il s'agit d'une entité qui encapsule des données (son état) et un comportement (des opérations sur l'état). Un objet est identifié de façon unique, et plusieurs instances, ou copies, d'un même objet peuvent exister. Les instances sont créées à partir d'un plan appelé classe ou bien à partir d'un autre objet qui est alors utilisé comme prototype des instances. L'accès à l'état d'un objet doit théoriquement se faire de manière exclusive à travers un ensemble de méthodes définies dans la classe de l'objet. » [Cer04]

Un second concept est central dans l'approche orientée objet : l'héritage. L'héritage permet à une classe appelée sous-classe d'obtenir les structures de données et comportements

d'une autre classe appelée classe de base ou super-classe. L'objectif de l'héritage est de permettre à la sous-classe d'étendre, de spécialiser ou de redéfinir le comportement de la super-classe. Outre une factorisation du code commun entre plusieurs classes, l'héritage permet aussi de mettre en œuvre le polymorphisme. Le polymorphisme permet de choisir à l'exécution une méthode à invoquer en fonction de la nature réelle des instances (et pas forcément du type spécifié syntaxiquement). De cette manière il est possible de conserver un code générique (en ne faisant référence qu'à des super-classes) alors qu'à l'exécution, des instances de sous-classes seront utilisées ; les appels aux méthodes redéfinies utiliseront alors les méthodes des sous-classes.

Les mécanismes d'adaptations qui tirent parti des objets ont massivement recours au polymorphisme. Nous allons ici présenter deux patrons de conception bien connus issus de [GHJV95] : le patron stratégie et la fabrique abstraite.

2.2.2.2 Le patron de conception stratégie

Le patron de conception *stratégie* offre la possibilité de changer dynamiquement d'algorithme pour effectuer un traitement. La figure 2.3 montre le diagramme de classes de ce patron. Dans notre exemple, l'interface *Stratégie* définit une méthode *strategie()*. Celle-ci donne lieu à un ensemble d'implantations dans des sous-classes. La sélection de la stratégie se fait dans la classe *contexte*, celle-ci maintient une référence vers l'une ou l'autre des stratégies, grâce au polymorphisme introduit par la hiérarchie de classes des stratégies.

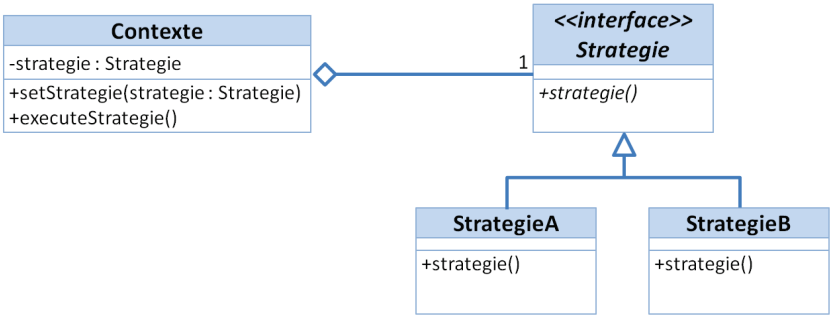

FIGURE 2.3 – Patron de conception stratégie.

Le mécanisme d'adaptation se situe dans la mise à jour de la référence, qui peut pointer vers l'une ou l'autre des stratégies en fonction des circonstances.

2.2.2.3 Le patron de conception pont

Le patron de conception *pont* a pour objectif de découpler une abstraction de son implantation pour permettre aux deux d'évoluer de manière indépendante [GHJV95].

La figure 2.4 montre que ce patron est divisé en deux parties. D'un côté, l'abstraction offre une interface standard pour utiliser un ensemble de fonctionnalités. Cette abstraction peut être raffinée en une ou plusieurs sous-classes, qui permettent de faire varier l'interface de programmation. D'un autre côté, une ou plusieurs implantations peuvent être fournies,

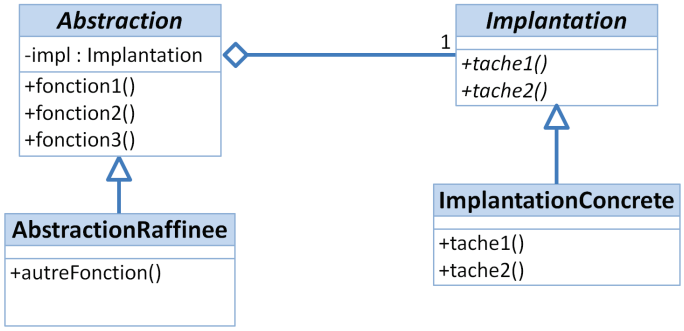

FIGURE 2.4 – Patron de conception pont.

comme dans le cadre du patron stratégie présenté précédemment. Par un lien maintenu entre l'abstraction et une implantation spécifique, l'abstraction peut fournir ses différentes fonctions en utilisant l'implantation.

Le mécanisme d'adaptation est donc double : il concerne aussi bien l'abstraction que l'implantation. Cependant, contrairement au patron stratégie dont le but est de faire varier un algorithme à l'exécution, ce patron est davantage orienté vers la conception. Il permet de garder un degré de flexibilité, afin de simplifier la mise à jour du code source. C'est pour cette raison qu'il n'est souvent utilisé qu'avec une seule implantation et une seule abstraction raffinée.

2.2.2.4 Limites de l'adaptation au niveau objet

Les patrons de conception proposent de structurer un ensemble de classes pour répondre à des problèmes récurrents. Ils permettent ainsi de se diriger plus rapidement vers des solutions éprouvées, tout en laissant la possibilité au programmeur d'adapter la conception aux besoins spécifiques de chaque situation.

Cependant, l'utilisation systématique des patrons de conception se heurte à un élément humain. On estime à dix mille le nombre de lignes qu'un développeur peut parfaitement maîtriser à un instant donné. Dans un contexte où les applications ont un nombre de lignes dépassant couramment plusieurs centaines de milliers, ce niveau d'abstraction se révèle très rapidement insuffisant afin d'adapter rapidement et fréquemment un logiciel.

Pour cette raison, il est apparu nécessaire de monter en abstraction, afin de permettre l'adaptation des applications à partir d'un plus faible nombre d'éléments mais de plus grande taille : les composants.

2.2.3 Adaptation au niveau composant

2.2.3.1 Définitions

Au cœur de l'approche à composants se trouvent trois concepts en relation, présentés ici par [HC01].

> "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."
>
> "A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model."
>
> "A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications."
>
> "These definitions demonstrate the important relationship between a software component infrastructure, software components, and a component model."

Nous pouvons compléter la définition de composant par les définitions suivantes :

> "A component is a static abstraction with plugs." [ND95]
>
> "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 party." [Szy97]

Un composant a donc les caractéristiques suivantes :

- c'est une **unité**, qui peut être boîte noire (si l'implantation est masquée) ou boîte blanche (si l'implantation est visible depuis l'extérieur) et qui possède dans tous les cas un contour clairement défini ;
- il possède un ensemble d'**interfaces requises et fournies**, définies formellement, qui permettent son utilisation (et sa réutilisation) sans connaissance de sa structure interne ;
- il est **exécutable** et peut être déployé (faisant éventuellement partie d'une composition) sur une infrastructure à composants (aussi appelée plate-forme à composants), sans apporter de modifications ;
- il est conçu conformément à un **modèle à composants**.

Poursuivons à présent avec la définition de modèle à composants, qui peut être complétée par la suivante :

> "The component model specifies the common rules that all developers must follow, e.g., basic requirements for the classification of elements as components, and certain patterns for assembling components." [ACF+07]

Un modèle à composants est donc un ensemble de règles pour implanter des composants, les assembler et permettre leur communication.

Enfin, la notion de plate-forme (ou infrastructure) à composants est le système logiciel qui exécute un ou plusieurs composants en interaction, conformément aux règles définies dans un modèle à composants.

> "A component-based system identifies 1) components, 2) an underlying platform, and 3) the binding mechanisms" [CSVC11]

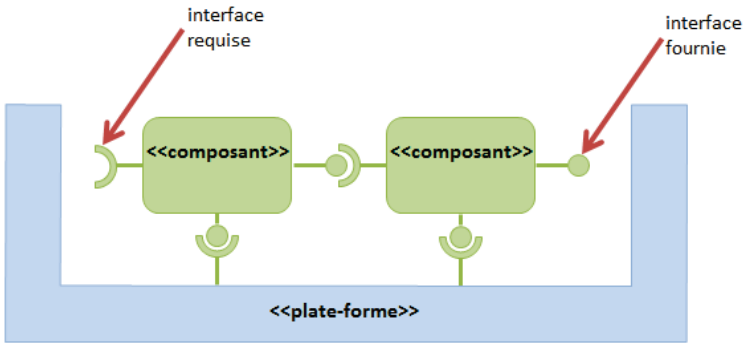

FIGURE 2.5 – Un système à composants [CSVC11].

La figure 2.5, adaptée de [CSVC11] récapitule ce que nous venons de présenter.

2.2.3.2 Techniques d'adaptation au niveau composant

Les techniques d'adaptation au niveau composant reposent sur la sélection, la substitution et la configuration de composants. La configuration n'étant pas spécifique aux composants, nous allons nous focaliser ici sur la sélection et la substitution.

L'adaptation au niveau composant consiste à modifier les relations entre ceux-ci, ce qui a pour conséquence de modifier l'assemblage global et donc l'application (figure 2.6). Ces modifications peuvent intervenir tout au long du cycle de vie du logiciel. Par exemple, lors de la compilation, il est possible de lier une implantation spécifique pour un composant. Cette technique est très facile à utiliser et offre souvent une garantie de fonctionnement ; puisque

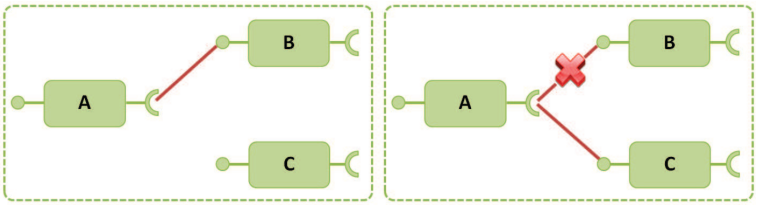

FIGURE 2.6 – Substitution de composant [LDM13].

toutes les ressources nécessaires à l'application sont connues avant l'exécution. Il est aussi possible d'effectuer une sélection de composants à partir d'une liste. Cette technique peut être utilisée pour créer des produits différents, en sélectionnant les fonctionnalités que l'on souhaite inclure au moment de la compilation. Par exemple, un logiciel permettant la supervision d'un *data center* pourrait prendre en charge plus ou moins de systèmes de virtualisation, selon le coût payé par le client. A l'autre bout du cycle de vie, la sélection dynamique d'implantation propose une modification lors de l'exécution. On dispose alors de plusieurs implantations pour une même fonctionnalité. Le choix de l'implantation utilisée est décidé lors de l'exécution et il peut éventuellement être modifié sans arrêt du système. Pour cela, le patron de conception stratégie [GHJV95], ou une approche orientée service peuvent par exemple être utilisés (paragraphe 2.3.2 page 27). Notons que cette technique est connue de longue date [Fab76], même si les concepts ont été clairement définis plus récemment.

Par rapport à l'adaptation au niveau code, le grain de substitution est donc largement plus conséquent et plus facile à manipuler. En effet, en tant qu'entité close et appréhendable par l'intermédiaire de ses interfaces requises et fournies, le composant est plus facilement compréhensible que l'ensemble du code de son implantation, qui peut comporter un nombre très conséquent de classes ou de paquets. Cependant, pour assembler les composants en une application cohérente, il est nécessaire de disposer d'un schéma directeur qui présente un ensemble de règles afin de guider la composition. Ces règles de haut niveau indispensables ne sont exprimées ni au niveau du code, ni au niveau des composants, mais à un troisième niveau que nous allons détailler à présent : l'architecture.

2.2.4 Adaptation au niveau architecture logicielle

2.2.4.1 Définition

Les définitions suivantes présentent une architecture logicielle (que nous qualifierons par la suite uniquement d'architecture) comme un ensemble de décisions de conception qui permettent de raisonner en l'absence du système considéré et qui contient des éléments, des relations et des propriétés :

> "A system architecture is the set of principal design decisions made during its development and any subsequent evolution." [MT10]

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." [BCK12]

De plus, en considérant plusieurs définitions de la notion de modèle, nous pouvons constater qu'une architecture est un modèle du système considéré. En effet, celle-ci permet de raisonner à propos du système original sans recourir directement à celui-ci. Cela n'est possible que sur un nombre limité de points de vue ; puisque l'architecture ne peut présenter la même complexité (et donc la même quantité d'informations) que le système original.

"A model is an abstraction of a physical system, with a certain purpose. It describes the physical system from a specific viewpoint and at a certain level of abstraction." [Abd00]

"A model is a set of statements about some system under study." [Sei03]

"A model is a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system." [BG01]

Les architectures sont étudiées depuis longtemps et largement utilisées pour les bénéfices qu'elles apportent :

- **une meilleure compréhension du système dans son ensemble** [SG96] en offrant une vision de plus haut niveau que le code. Cet atout est qualifié d'extrêmement précieux par [MN97], et même de besoin critique par [LM08] dès que le système est pris en charge par plus d'un développeur. De plus, lors de la phase de maintenance, le passage par l'architecture permet une compréhension (et donc une correction) plus rapide du logiciel [GTTGPn+11].
- **une communication plus aisée entre les différents acteurs** en se focalisant sur une compréhension commune de haut niveau ;
- **un support éventuel pour des vérifications formelles** [Luc96, ADG98].

Malheureusement, dans de très nombreux cas, même s'ils sont initialement conformes avec leur architecture, les logiciels ont tendance à évoluer indépendamment de celle-ci [DL12]. L'architecture ne reflète plus le logiciel et devient alors inutile. Pour cette raison de nombreux travaux se sont intéressés depuis longtemps à l'extraction d'architecture [MN97, LM08].

Cependant, même si ces approches parviennent à obtenir des résultats intéressants, la grande complexité de ce travail automatisé de rétro-ingénierie ne permet pas d'obtenir des résultats aussi satisfaisants que ceux qui auraient été obtenus « à la main », en raison de la complexité à distinguer les informations qui doivent faire partie du modèle des autres. Pour cette raison, [GS04] souligne que sans une forme de garantie de cohérence, la relation entre l'architecture et l'implémantation est hypothétique et beaucoup des avantages liés à l'utilisation d'architectures sont perdus.

2.2.4.2 Adaptation au niveau architecture logicielle

L'adaptation au niveau architectural propose de dépasser la vision proposée par les composants tout en s'appuyant dessus. Avec ce troisième niveau, il devient possible d'adapter très fortement l'application, en modifiant sa structure tout en validant la cohérence de celle-ci. L'ajout et la suppression de composants, ainsi que la modification des liaisons entre ceux-ci sont donc effectués avec un niveau de contrôle supplémentaire à celui fourni par les seuls composants. Cette idée d'utiliser des architectures pour l'adaptation (et même l'auto-adaptation, dont nous parlerons au chapitre suivant) n'est pas récente. De plus, en l'espace d'une décennie, les motivations sont restées sensiblement les mêmes [OGT+99, GSC09].

Tout d'abord, en tant que modèle, les architectures offrent une perspective globale sur le système considéré et mettent en valeur les comportements et les propriétés clés. Les architectures offrent ainsi une hauteur de vue et une focalisation sur les aspects les plus importants. Elles facilitent ainsi la compréhension globale du système, indispensable avant de modifier celui-ci pour effectuer une adaptation.

De plus, en tant qu'expression des décisions de conception, les architectures permettent de formaliser les contraintes du système, et ainsi, de vérifier la validité d'un changement. Cette propriété est particulièrement intéressante puisque la taille des systèmes augmentant, il est de plus en plus difficile pour un développeur de prévoir les conséquences d'une modification, même localisée. Cependant, il convient de noter que les vérifications que permettent les modèles sont effectuées uniquement dans le champ couvert par ceux-ci : la validité d'une modification n'est vérifiée qu'en fonction du jeu de contraintes formalisées dans l'architecture.

Malgré ces bénéfices identifiés de longue date, nous devons constater que l'adaptation au niveau architectural est encore très largement sous-étudiée. C'est ce qui ressort de la conférence VARSA 2011, dont l'objet est la variabilité dans les architectures logicielles. Un compte rendu de la conférence [GAWM11] apporte un éclairage particulièrement intéressant. Il pointe que la variabilité, nécessaire à l'adaptation, est souvent non décrite explicitement dans les architectures logicielles, quand celles-ci sont décrites en dehors des lignes de produits. Dans de nombreux cas, une connaissance existe dans l'esprit des architectes, mais elle reste à l'état de connaissance implicite. Selon certains participants, il est tout simplement trop complexe de modéliser toute la variabilité dans l'architecture. De plus, de nouvelles approches pour la variabilité ne peuvent pas être utilisées, en raison de l'absence d'outillage pour la gérer convenablement.

De notre point de vue, cela explique le très faible nombre de contributions sur l'adaptation au niveau architectural, par rapport à ce qui est disponible, notamment au niveau du code. Par exemple, les différentes techniques décrites par [BC13] pour porter l'adaptation au niveau architectural sont une simple extension de celles décrites pour le niveau composant. Il manque clairement une vision qui permette de comprendre l'architecture de manière plus globale, au-delà d'un un simple ensemble de composants liés entre eux.

2.2.5 Synthèse

L'adaptation est une tâche à la fois nécessaire et complexe. Nécessaire, pour répondre à de nouveaux besoins, aux changements de disponibilité des ressources, à la nécessité de corriger des dysfonctionnement, et à la prise en compte de la mobilité des utilisateurs [Gar13]. Complexe, parce qu'elle peut avoir des conséquences sur le logiciel en de nombreux points, et nécessite à la fois une excellente compréhension du logiciel et une technicité importante.

La mise en œuvre de ces adaptations peut être effectuée à plusieurs niveaux d'abstraction. Au niveau code, au niveau objet, au niveau composant et au niveau architecture. Si les adaptations au niveau code semblent les plus immédiates à mettre en œuvre, elles sont aussi extrêmement complexes et demandent une technicité importante, d'autant plus quand le logiciel est impacté de manière transversale. Pour simplifier l'écriture et fiabiliser le code produit, les patrons de conception de l'approche orientée objet proposent des modèles éprouvés pour guider l'implantation des mécanismes d'adaptation. Cependant, l'utilisation systématique des patrons de conception se heurte à un élément humain. Le niveau composant permet de dépasser cette limite en apportant un grain de taille plus importante, et donc mieux adapté à la manipulation d'applications de grande taille. Il peut donc pallier les insuffisances mises en avant au niveau objet.

Le niveau architectural prolonge le niveau composant en plaçant au cœur de ses préoccupations la mise à jour de la topologie de l'application. Au-delà de la substitution d'un composant par un autre, prise en charge au niveau composant, le niveau architectural permet de modifier des composants et les relations qu'ils entretiennent tout en contrôlant les conséquences des changements et en vérifiant que les modifications apportées restent conformes avec les décisions de conception. Ce domaine de recherche est encore en plein développement [GAWM11]. Même s'il reste encore beaucoup de travail à accomplir, des progrès ont été effectués en l'espace d'une décennie [CHG+04]. Ceux-ci ont pour la plupart tiré parti de patrons architecturaux [AAG93, BCK12] qui offrent des solutions éprouvées à des problèmes récurrents. Par exemple, le style architectural « pipe-and-filter » [BCK12] offre une approche claire et structurée pour transformer ou filtrer de manière incrémentale un flux de données. Le patron « publish-subscribe » [BCK12] permet quand à lui à des systèmes de communiquer par envoi de messages en fiabilisant la transmission de ceux-ci.

L'un de ces patrons architecturaux nous apparaît très intéressant dans le cadre de notre problématique d'adaptation. Il s'agit de l'approche à services, qui permet de mettre en œuvre des stratégies d'adaptation, notamment grâce à un couplage large entre le producteur et le consommateur de service. Nous allons donc dans le chapitre suivant présenter cette approche qui permet l'implantation d'applications de grande taille avec un point de vue architectural.

2.3 Services pour l'adaptation au niveau architectural

Les services sont des entités auto-suffisantes et faiblement couplées. Ils sont aujourd'hui largement utilisés pour implanter des mécanismes d'adaptation au niveau architectural. Pour expliquer les raisons de ce succès, nous allons tout d'abord présenter les services. Nous verrons ensuite les différents acteurs en jeu lors de leur utilisation dans le cadre de l'approche orientée service. Ensuite, nous élargirons notre perspective à la réalisation des applications qui utilisent cette approche avec les architectures orientées services, puis les composants orientés service. Enfin, nous discuterons des avantages et des inconvénients de l'utilisation des services pour l'adaptation au niveau architectural.

2.3.1 Notion de service

La notion de service est complexe, et elle recouvre une multitude d'aspects différents. Nous allons tout d'abord détailler ces caractéristiques. Ensuite, nous donnerons une définition, qui permet de reprendre les critères les plus importants.

Tout d'abord, un service peut être considéré comme une **boîte noire**, qui n'expose que les informations nécessaires à sa sélection et son invocation. Les clients peuvent donc l'utiliser sans connaître sa technologie d'implantation ni sa plate-forme d'exécution [Pap03]. Symétriquement, un service est utilisé sans avoir connaissance du contexte dans lequel le client va faire appel à lui. C'est pour cela que l'on parle d'**indépendance à double sens** [Cho09].

De plus, un service est une **entité auto-suffisante**, qui peut délivrer à elle seule une fonctionnalité métier [PH07]. Cette propriété permet aux services d'être facilement utilisables par différents consommateurs. Les services sont donc **réutilisables**. Cela est d'ailleurs l'une des raisons majeures du succès de ceux-ci.

Pour exposer à ses clients potentiels ses fonctionnalités, un service est décrit dans une **description de service** [Ars04]. Cette connaissance est une information nécessaire et suffisante pour qu'un consommateur puisse l'utiliser. Cela permet de réduire significativement la quantité de données échangées entre le consommateur et le fournisseur de service. En particulier, aucune information concernant la plate-forme d'exécution ou la technologie d'implantation du service n'est échangée. Notons que les services sont aussi indépendants de l'état et du contexte d'exécution des autres services [PH07]. Enfin, un service peut être **soit local, soit distant** [Sim11].

Compte tenu des multiples caractéristiques que présentent les services, il existe de nombreuses définitions qui mettent l'accent sur des caractéristiques différentes [Pap03, Ars04, PTD+06, PH07, MLM+06].

Nous proposons ci-dessous la définition de S. Chollet dans [Cho09], qui reprend la majorité des points mentionnés ci-dessus.

« [Un service est] une entité logicielle qui fournit un ensemble de fonctionnalités définies dans une description de service. Cette description comporte des informations sur la partie fonctionnelle du service mais aussi sur ses aspects non-fonctionnels. A partir de cette spécification, un consommateur de service peut rechercher un service qui correspond à ses besoins, le sélectionner et l'invoquer en respectant le contrat qui a été accepté par les deux parties. » [Cho09]

2.3.2 Approche orientée service

2.3.2.1 Principe d'interaction de base

Pour utiliser de manière souple les services dans le cadre d'une application, l'approche orientée service, en anglais *Service-Oriented Computing* (SOC), définit un style architectural que nous allons à présent détailler. Cette approche a été largement popularisée par l'une des technologies qui utilise ce paradigme : les services Web [CDK+02]. Il y a par ailleurs de nombreuses autres technologies qui suivent l'approche orientée service, comme par exemple Jini [ASW+99], UPnP [UPn08] utilisé dans le contexte des services répartis, ou encore OSGi™ [All09] pour les services localisés sur une même machine virtuelle Java™.

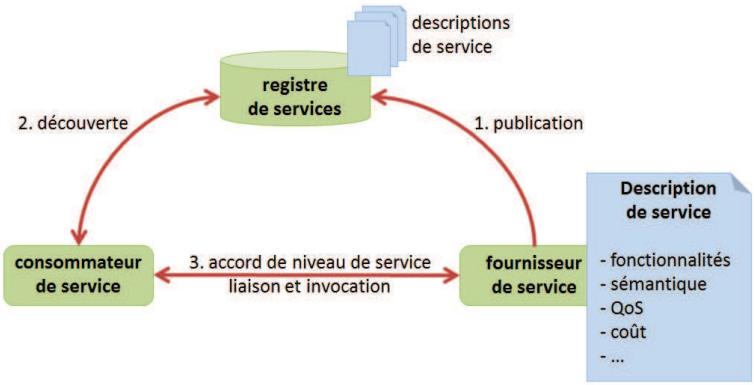

FIGURE 2.7 – Acteurs et interactions dans l'architecture à services [Pap03].

Nous allons nous détacher des implantations particulières, pour introduire ce style architectural, en présentant tout d'abord les trois acteurs sur lesquels il repose (figure 2.7) :

- **le fournisseur de service**, qui propose un service décrit dans une description de service ;
- **le consommateur de service**, qui utilise un ou plusieurs services ;
- **le registre de services** (aussi appelé annuaire, ou courtier de services), qui stocke un ensemble de descriptions de services pouvant provenir de plusieurs fournisseurs différents. Il offre deux interfaces principales : l'une est dédiée aux fournisseurs de services, afin de permettre l'enregistrement de descriptions de services et l'autre aux consommateurs de services, pour rechercher les services et parcourir les descriptions.

Ces trois acteurs sont en interaction en utilisant trois primitives de communication (figure 2.7) :

- **la publication de service dans l'annuaire** permet à un fournisseur de service de demander l'enregistrement d'une description de service dans l'annuaire ;
- **la découverte de service dans l'annuaire** offre la possibilité à un consommateur de services de récupérer dans l'annuaire l'ensemble des descriptions de service correspondant à une fonctionnalité souhaitée ;
- **la liaison et l'invocation d'un service** permet au consommateur de service d'utiliser un service. Cette étape comprend aussi la négociation de l'accord de niveau de service, détaillée au paragraphe suivant.

2.3.2.2 Accord de niveau de service

Avant d'invoquer le service, le consommateur et le fournisseur négocient un accord de niveau de service (en anglais, *Service Level Agreement*, SLA) [AFM05], clarifiant ainsi leurs attentes et engagements réciproques [Mil87] concernant la qualité de la réalisation du service.

> « Un accord de niveau de service est un contrat entre au moins deux parties, dont les clauses portent sur la qualité du (ou des) service(s) faisant l'objet du contrat. » [Tou10]

Dans le cas où les engagements ne sont pas satisfaits, des pénalités peuvent être appliquées, elles aussi le fruit d'une négociation préalable.

Les objectifs de l'accord de niveau de service ont été classés selon quatre niveaux [BJPW99]. Chaque niveau assure la qualité du niveau précédent, et ajoute de nouvelles contraintes :

- **le niveau de base :** il porte simplement sur les opérations, leurs entrées et sorties, ainsi que les éventuelles exceptions pouvant se produire lors du traitement. Ces informations peuvent être décrites au niveau de la signature des méthodes ou dans un langage de description d'interfaces [3].
- **le niveau comportemental :** il introduit des assertions booléennes, appelées pré et post conditions pour tous les services, ainsi que pour les classes invariantes. Cette approche a été pour la première fois implantée sous le nom de conception par contrat dans le langage Eiffel [Mey92].
- **le niveau de synchronisation :** comme les services ne sont pas toujours atomiques ou transactionnels, ce niveau permet de spécifier un ordonnancement entre les appels de méthodes du service. Ceux-ci peuvent être en séquence, en parallèle ou sans contraintes.

3. En anglais, *Interface Description Language* (IDL).

- **le niveau qualité de service :** ce dernier niveau définit la qualité attendue de la livraison du service. Par exemple, il est possible de spécifier le temps maximum ou moyen de réponse, la précision du résultat obtenu... Notons qu'il faut pour cela que la qualité du service soit mesurable avec, par exemple, une unité de temps (pour les délais) ou un taux (pour la précision), spécifié par un objectif de niveau de service (en anglais, *Service Level Objectives*, SLO).

La gestion de cet accord est effectuée dans le cadre plus large de la gestion du niveau de service (en anglais, *Service Level Management*, SLM). Ce processus inclut plusieurs tâches [WWW+06] :

- la négociation permettant d'aboutir à la définition d'un accord de niveau de service ;
- la supervision, afin de mesurer les différents niveaux de qualité de service et vérifier ainsi leur conformité avec l'accord de niveau de service ;
- si nécessaire, la gestion de mécanismes d'adaptation préventifs ou correctifs afin de maintenir la qualité de service au niveau souhaité.

2.3.2.3 Approche orientée service dynamique

Jusqu'alors, nous n'avons traité que de d'établissement de liaisons avec les services. Or, il peut aussi arriver qu'un service disparaisse ou qu'un nouveau plus adapté aux besoins du consommateur apparaisse. L'objectif de l'approche orientée service dynamique est de prendre en compte ces autres cas de figure [Esc08]. Pour cela, deux nouvelles primitives sont ajoutées au registre de services :

- **la primitive de retrait de service** permet au fournisseur de service de notifier le registre de services de son impossibilité à continuer à délivrer un service jusqu'alors fourni ;
- **la primitive de notification** permet au registre de services d'informer les consommateurs du départ ou de l'arrivée d'un service. Cela permet ainsi aux consommateurs de s'ajuster au plus tôt à la disponibilité fluctuante des services.

La figure 2.8 illustre les étapes d'interaction nécessaires afin de prendre en compte l'arrivée d'un service et la figure 2.9 son départ.

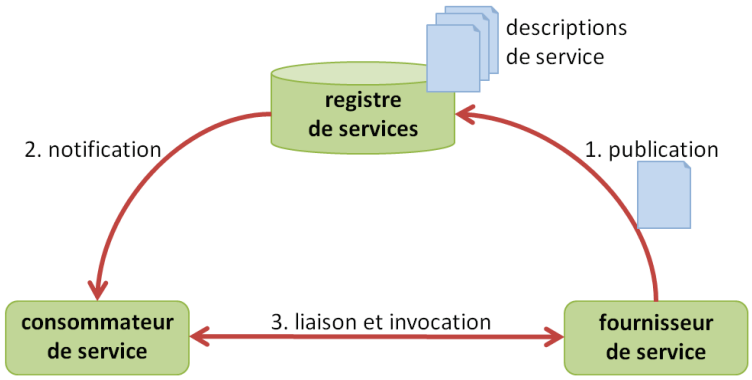

FIGURE 2.8 – SOC dynamique – arrivée d'un service.

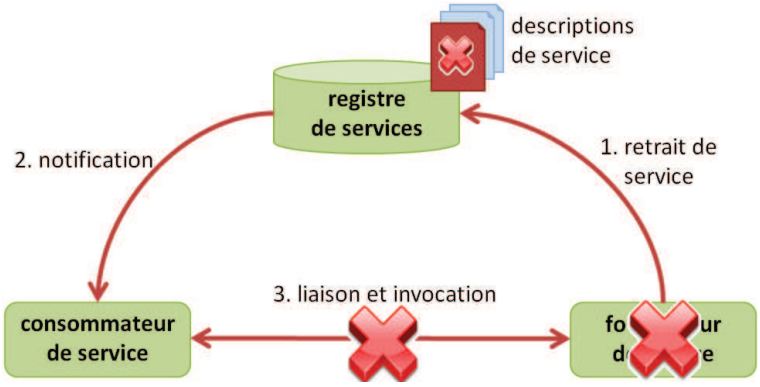

FIGURE 2.9 – SOC dynamique – départ d'un service.

2.3.2.4 Bénéfices de l'approche orientée service

Grâce à l'approche orientée service, il est possible à un consommateur d'utiliser les services d'un fournisseur sans le connaître a priori. De même, il est possible à un fournisseur de proposer ses services à des clients non identifiés à l'avance. Pour cette raison, on dit que l'approche orientée services permet un **faible couplage** [PH07] entre le fournisseur et le consommateur, grâce au registre de services qui permet leur mise en relation sur la base de la description de service.

Le faible couplage permet aussi la **substituabilité**. Toujours grâce au registre de services, le consommateur peut décider de changer de fournisseur de services. Cela est d'ailleurs facilité par les primitives ajoutées dans le cadre de l'approche orientée services dynamique.

Le faible couplage offre encore un autre avantage : il accepte le **masquage de l'implantation du service**. En conséquence, l'utilisation d'un service peut être moins dépendante d'une technologie particulière et sa réutilisabilité s'en trouve ainsi accrue. De plus, comme il n'est pas nécessaire de connaître les détails d'implantation du service pour l'utiliser, cela facilite la compréhension de ses capacités, qui doivent être exprimées uniquement dans sa description.

Les applications doivent évoluer de plus en plus rapidement, en raison de l'évolution rapide des besoins, de la mise en place de nouvelles organisations dans le cadre de restructurations, de fusions ou d'acquisitions. En offrant une description abstraite et de haut niveau des services, l'approche orientée service se révèle bien adaptée à ces recombinaisons fréquentes. Leur utilisation permet donc d'**accroître l'agilité globale du système d'information**.

L'approche orientée service peut aussi **favoriser la distribution** de l'application. En effet, le consommateur ne connaît pas à l'avance la localisation du fournisseur de service, qui peut être local ou distant.

2.3.3 Architecture orientée service

2.3.3.1 Architecture de base

Nous venons de détailler un style architectural : l'approche orientée service. Une architecture orientée service, en anglais *Service-Oriented Architecture* (SOA), propose un ensemble de technologies permettant la mise en place des applications suivant le paradigme de l'approche orientée service [GP08]. C'est donc un intergiciel qui permet aux applications de fournir, publier, découvrir et utiliser des services. Les mécanismes à l'œuvre dans le cadre d'une architecture orientée service sont répartis selon deux catégories [EAA+04] (figure 2.10) :

- **les mécanismes de base** implantent les mécanismes de base de l'approche orientée service, décrits précédemment : la publication, la découverte, la composition, la contractualisation et l'invocation des services ;
- **les mécanismes étendus** implantent un ensemble spécifique de propriétés non-fonctionnelles afin de prendre en charge ses contraintes propres, telles que la sécurité, les transactions, l'administration, etc.

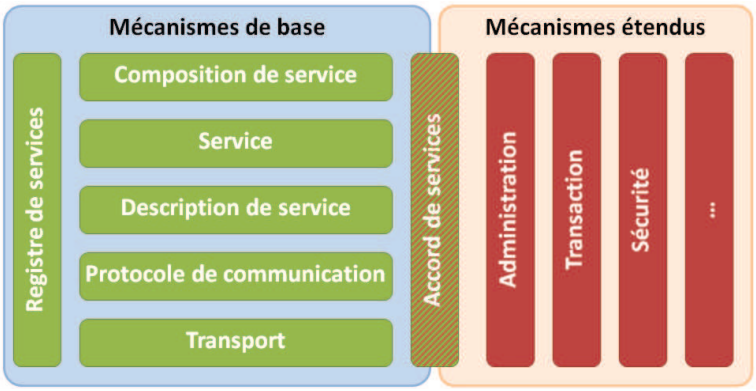

FIGURE 2.10 – Architecture orientée service : mécanismes d'implantation [EAA+04].

Les mécanismes de base sont présents dans toutes les technologies d'architecture orientée service, alors que les mécanismes étendus sont optionnels et implantés selon l'objectif et les contraintes de l'application.

2.3.3.2 Architecture orientée service étendue

Les architectures orientées service utilisent l'approche à services pour bâtir des applications. Cependant, elles manquent d'un point de vue de plus haut niveau pour traiter la problématique de composition et de gestion nécessaire à la création d'applications de grande taille. C'est là l'objectif des architectures orientées service étendues (en anglais, xSOA pour *extended SOA*), une extension aux architectures orientées service.

Dans [PH07], M. P. Papazoglou présente les architectures orientées service étendues sous la forme d'une pyramide à trois niveaux (figure 2.11), chaque niveau s'appuyant sur les fonctionnalités offertes par les niveaux inférieurs :

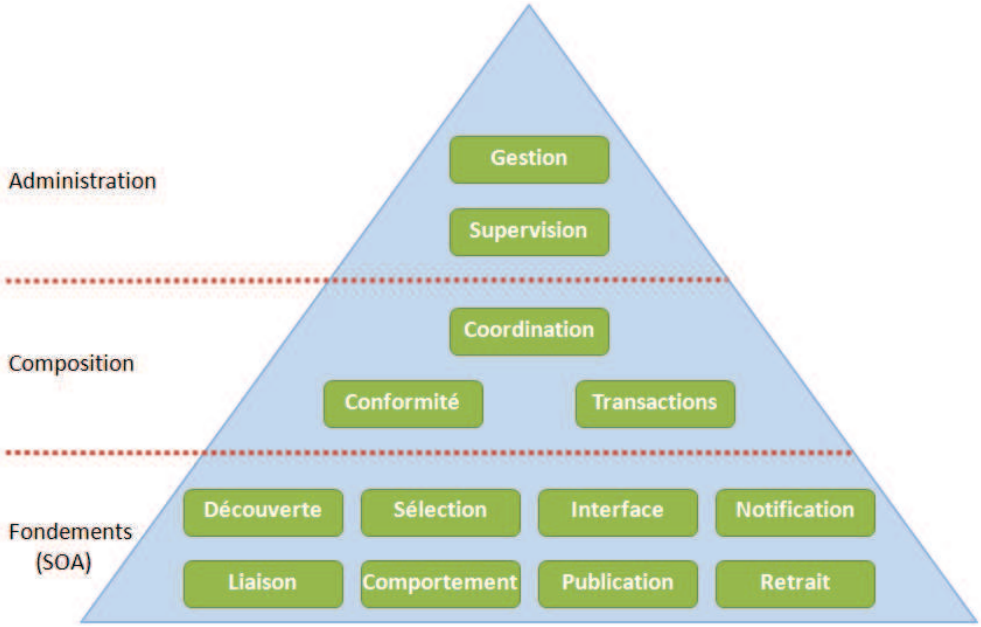

FIGURE 2.11 – Pyramide des fonctionnalités des architectures orientées service étendues (à partir de [PH07]).

- le premier niveau propose les fonctionnalités offertes dans le cadre des architectures à services. Il s'agit donc de l'infrastructure d'exécution qui utilise des services ;
- le second niveau offre la capacité d'agrégation de plusieurs services en un unique service composite. Celui-ci peut ensuite être utilisé comme un service ordinaire, en étant publié dans le registre de services ;
- enfin, le dernier niveau est consacré à la gestion des services. Cela passe par la définition et la surveillance de la qualité de service, l'audit en continu de la plate-forme d'exécution, la résolution des erreurs, l'allocation et la libération des ressources matérielles et logicielles, etc.

2.3.3.3 Architecture orientée service étendue dynamique

Si l'architecture proposée par M. P. Papazoglou se révèle extrêmement structurante, elle souffre néanmoins d'une limitation importante : elle ne permet pas de prendre en charge le retrait de service (et donc aussi la notification) que nous avons présenté dans l'approche orientée service dynamique (paragraphe 2.3.2.3 page 29).

C. Escoffier a cherché à dépasser cette limite en proposant la notion d'architecture orientée service étendue dynamique [Esc08]. Avec celle-ci, le départ des services est pris en charge lors de l'exécution et la composition devient pleinement dynamique. Pour cela, il est nécessaire de modifier la pyramide de M. P. Papazoglou en ajoutant des fonctionnalités à chaque niveau :

- le premier niveau doit être basé sur l'approche orientée service dynamique. Les fonctionnalités de retrait et de notification doivent donc être présentes.

- le second niveau, centré sur la composition doit pouvoir tirer parti du retrait de service et assurer notamment la substitution de ceux-ci lors de l'exécution ;
- le dernier niveau doit être en mesure de vérifier en permanence la cohérence du système en fonction des objectifs de l'application.

De manière transversale, le dynamisme a aussi des conséquences sur les propriétés non-fonctionnelles (telles que la sécurité) qui deviennent plus complexes à garantir dans un contexte où l'application peut dépendre d'implantations de services différents au cours du temps.

2.3.4 Approche à composants à services

2.3.4.1 Définition

Le rapprochement de l'approche à services avec l'approche à composants a donné naissance à l'approche à composants à services [Cer04]. L'objectif de cette nouvelle approche est de faciliter la construction d'applications et leur adaptation à l'exécution en utilisant l'approche à services pour résoudre les dépendances entre les composants [TLR+09]. Cela permet de retarder la sélection des implantations jusqu'à l'exécution et de réagir à l'arrivée ou au départ de celles-ci. Un modèle à composant orienté services est basé sur les principes suivants [CH04] :

- **un service fournit des fonctionnalités :** un service propose une ou plusieurs opérations réutilisables ;
- **un service est décrit par une spécification de service :** celle-ci contient les informations nécessaires pour son invocation et sa découverte en décrivant la syntaxe, le comportement et la sémantique du service ;
- **un composant implante une spécification de service :** le composant doit donc respecter les contraintes qui ont été définies dans la spécification. Le service peut aussi déclarer d'autres dépendances de services rendues nécessaires par sa technologie d'implantation ;
- **le modèle d'interactions de l'approche à services dynamique est utilisé pour résoudre les dépendances de services :** les services fournis par les instances de composants ont leur description publiée dans le registre de services. Celui-ci est utilisé pour résoudre les dépendances entre composants à l'exécution ;
- **les compositions sont décrites sous forme de spécifications de services :** la composition est un ensemble de spécifications de services qui permet de choisir les composants à instancier. Les liaisons entre composants sont établies lors de l'exécution et n'ont pas à être déclarées statiquement ;
- **les spécifications de services sont les fondements de la substituabilité :** dans une composition, un composant peut être remplacé par un autre composant qui respecte la même spécification de service.

L'approche à composants à services permet donc de combiner les avantages :

- **de l'approche à composants :** un modèle de développement simple, une description de la composition ;
- **de l'approche à services dynamique :** un faible couplage et l'établissement dynamique des liaisons, ce qui évite la définition statique de celle-ci.

Il existe de nombreux modèles à composants à services, comme par exemple Gravity [CH04], SCA [4] avec FraSCAti [SMF+09] qui en est une implantation, OSGi™ Declarative Services (basé sur ServiceBinder [CH03]) ou encore le conteneur Apache Aries Blueprint [5] (derivé de Spring Dynamic Modules [CTP10]). Nous allons présenter celui que nous avons utilisé dans le cadre de ces travaux de thèse : Apache Felix iPOJO [6], développé au sein de l'équipe Adèle du LIG [7] par Clément Escoffier.

2.3.4.2 Apache Felix iPOJO

Apache Felix iPOJO (pour *injected Plain Old Java Object*) [EHL07, Esc08] est une infrastructure de développement et d'exécution d'applications à base de composants à services. Elle a pour but de simplifier de manière extrêmement importante le développement d'applications sur la plate-forme OSGi™.

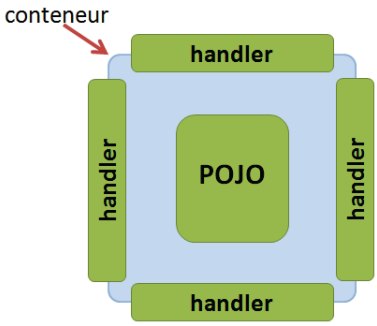

FIGURE 2.12 – Conteneur de Apache Felix iPOJO [EHL07].

Le *framework* iPOJO repose sur la notion de conteneur (figure 2.12). A l'intérieur de celui-ci, se trouve le code métier, sous forme d'objets Java habituels (POJO, *Plain Old Java Object*). C'est ce code qui est écrit par le développeur et injecté dans le conteneur lors de la phase de compilation. Cette dernière étape, très complexe, est totalement automatisée et transparente pour le programmeur.

Le *framework* iPOJO propose une séparation des préoccupations par l'intermédiaire de points d'extension appelés *handlers* qui peuvent être ajoutés au conteneur. Cela permet aux

4. En anglais *Service Component Architecture* `http://www.oasis-opencsa.org/`
5. `https://aries.apache.org/modules/blueprint.html`
6. `https://felix.apache.org/documentation/subprojects/apache-felix-ipojo.html`
7. Laboratoire d'Informatique de Grenoble.

développeurs de se focaliser sur le code métier, tout en déléguant à la plate-forme une série de préoccupations non-fonctionnelles.

Les *handlers* de base d'iPOJO permettent l'importation et l'exportation de services. Le *Service Providing Handler* permet ainsi au programmeur d'exporter simplement une fonctionnalité offerte par un composant sous forme d'un service qui devient alors accessible sur la plate-forme OSGi™ sous jacente. Symétriquement, le *Dependency Handler* permet la résolution automatisée des dépendances de services en prenant en charge la découverte et de l'injection dynamique des services au niveau du code métier. Ainsi, l'application peut s'adapter à l'arrivée ou au départ de services de manière automatique et transparente pour le programmeur. Ce double mécanisme, de mise à disposition de service d'une part, et d'injection de service requis d'autre part, permet la construction d'applications par simple déclaration de dépendances de services. Grâce à cela, nous avons l'établissement de liaisons dynamiques entre les composants et ainsi la construction automatisée d'applications.

Les deux *handlers* que nous venons de présenter étant à la base d'iPOJO, il est courant de représenter le conteneur selon le schéma de la figure 2.13. Ce schéma simplifié met l'accent sur les services fournis et requis qui sont gérés par les *handlers*, sans faire figurer directement ces derniers, ni mentionner les autres *handlers* qui peuvent éventuellement être présents.

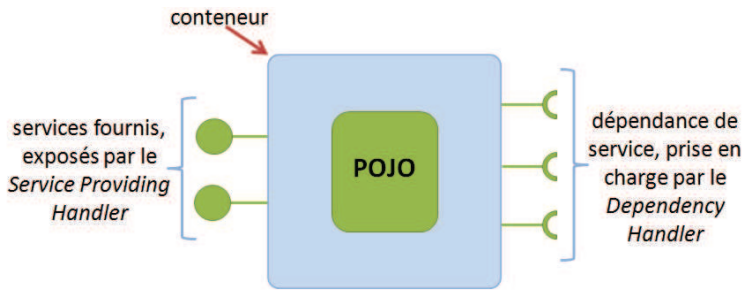

FIGURE 2.13 – Vue simplifiée du conteneur de Apache Felix iPOJO.

iPOJO est fourni avec tout un ensemble de *handlers* prêts à l'emploi et offre la possibilité de créer des *handlers* personnalisés. En plus des *handlers* de gestion de dépendances que nous venons de présenter, iPOJO fournit par exemple :

- **le *Event Admin handler***, qui fournit un mécanisme pour simplifier la communication entre les instances de composants iPOJO. Les composants peuvent ainsi émettre facilement des événements, sur des files de messages appelées *topics* ;
- **le *log handler***, qui offre un mécanisme simple et configurable de gestion des traces d'exécution ;
- **le *JMX handler***, qui permet d'ajouter des capacités d'administration en utilisant le protocole JMX [8] (*Java Management Extensions*).

iPOJO propose donc une implantation de composants orientés services au-dessus d'OSGi™. Pour cela, le *framework* définit un conteneur dans lequel des *handlers* peuvent être

8. http://www.oracle.com/technetwork/java/javamail/javamanagement-140525.html

ajoutés. Ceux-ci permettent notamment de prendre en charge la liaison dynamique entre les composants, grâce à l'approche à services.

De par son implantation sur la plate-forme OSGi™, iPOJO hérite de deux limitations de celle-ci : le code doit être exécuté sur une machine virtuelle Java™et son exécution est centralisée et locale [Sim11]. Une autre limitation d'iPOJO provient de l'approche à composants orientés services à proprement parler. L'établissement des liaisons est certes automatisé sur la base d'interfaces, mais en contre-partie, il n'existe pas de schéma directeur au niveau de l'application pour guider l'établissement de ces liaisons. Celles-ci ne sont établies qu'au niveau de chaque composant, sans « conscience » de l'architecture globale qui se dessine et se modifie à chaque modification de liaison.

2.3.5 Synthèse

Les architectures orientées services que nous venons de présenter permettent de bâtir des applications de grande taille avec le patron d'interactions à trois acteurs de l'approche orientée service. Cette approche offre un point de vue adapté pour comprendre l'application et l'adapter le cas échéant. En effet, le couplage large entre les consommateurs et les fournisseurs de services permet un découpage naturel de l'application. De plus la nature même de l'approche orientée service permet de mettre en œuvre de la flexibilité au niveau de la structure architecturale.

Si le dynamisme est pris en charge, il devient aussi possible d'adapter « à chaud » et de manière automatisée une application en changeant de fournisseur de service en s'adaptant au plus tôt à l'apparition ou à la disparition de services. En cela, l'approche à service facilite grandement l'adaptation dynamique des applications en tirant parti du niveau architectural.

L'approche à services a aussi permis la création des composants orientés services. Dans cette approche, les liaisons entre les composants peuvent être créées et modifiées à l'exécution. L'application peut ainsi se construire de manière automatisée, tout permettant au développeur de conserver un point de vue architectural sur l'application produite et mise à jour.

Cependant, si les paradigmes de l'approche orientée service et des composants orientés services permettent une grande souplesse, on peut désirer a contrario mieux maîtriser les architectures produites, en définissant en amont des contraintes supplémentaires pour guider la composition des applications. Cette préoccupation est centrale dans l'ingénierie des lignes de produits logicielles que nous allons étudier maintenant.

2.4 Lignes de produits logicielles

Avec les lignes de produits logicielles, un cadre extrêmement contrôlé permet de guider la conception d'architectures selon un ensemble de contraintes. Nous allons présenter cette approche en deux temps. D'abord, nous nous intéresserons aux lignes de produits logicielles classiques, qui proposent une adaptation uniquement lors de la conception. Nous exposerons ensuite les lignes de produits dynamiques, qui permettent de résoudre la variabilité des lignes de produits à l'exécution, permettant ainsi la gestion de l'adaptation à l'exécution.

2.4.1 Introduction

L'ingénierie des lignes de produits logicielles (appelée aussi famille de produits) trouve son inspiration dans les lignes de production industrielles. Elle vise à bâtir un processus permettant de créer rapidement et à moindre coût un ensemble de produits appartenant au même domaine à partir d'éléments réutilisables [CN02].

Pour introduire les lignes de produits logicielles, commençons par cette définition du Software Engineering Institute :

> "A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." `http://www.sei.cmu.edu/productlines/`

A travers cette définition, nous pouvons constater que le concept de réutilisabilité est central dans les lignes de produits. C'est lui qui donne corps à l'ensemble des systèmes logiciels qui la composent en rassemblant autour d'un ensemble de fonctionnalités des produits qui s'adressent à un segment de marché ou un but commun.

En pratique, les lignes de produits sont souvent organisées autour de deux processus [WL99, Yu10] que nous allons détailler à présent : l'ingénierie du domaine et l'ingénierie applicative.

2.4.2 L'ingénierie du domaine

2.4.2.1 Objectif de cette étape

L'objectif de cette première étape est de définir la portée de la ligne de produits ; c'est-à-dire l'ensemble des membres qui la composent.

> "Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised." [PBvdL05]

Pour cela, deux types d'éléments sont identifiés : ceux qui sont partagés entre tous les membres d'une ligne de produits (les *commonalities* de la définition précédente), et ceux qui sont spécifiques à un ou plusieurs membres (les *variabilities*). On va donc chercher à avoir une portée suffisamment grande pour tirer le plus largement possible parti de la réutilisation, mais pas trop grande non plus, afin de conserver une large base partagée entre les membres de la famille et de ne pas effectuer trop de développements spécifiques [Cle02].

On va ensuite définir de manière extrêmement fine les contraintes entre ces éléments (implication, exclusion, alternative). Cette tâche, centrale dans le cadre des lignes de produits, permet d'exprimer la variabilité entre les différents produits qui la composent.

Pour cela, plusieurs techniques ont été développées. Celles-ci peuvent être réparties selon deux groupes [SD07] : les approches basées sur les fonctionnalités et les approches basées sur les décisions. Pour introduire l'une et l'autre, nous allons décrire la technique la plus populaire pour chaque approche : FODA, pour la modélisation de la variabilité basée sur des fonctionnalités et le modèle de variabilité orthogonale (OVM, *Orthogonal Variability Model*) qui considère la variabilité en termes de décisions.

2.4.2.2 Diagramme de fonctionnalités d'après FODA

Les diagrammes de fonctionnalités ont été introduits par [KCH+90] pour la méthode FODA (*Feature-Oriented Domain Analysis*). Comme le nom l'indique, ils reposent sur le concept de fonctionnalité.

> "A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate between systems." [CHE04]

Une fonctionnalité correspond donc à un morceau d'application clairement identifié, qui fait sens et qui peut être utilisé dans un ou plusieurs produits de la ligne de produits. Toutes les fonctionnalités de la ligne de produits sont présentes dans le diagramme de fonctionnalités qui permet par une notation graphique d'indiquer pour chaque produit les fonctionnalités obligatoires (*commonalities*) et les fonctionnalités facultatives (*variabilities*). De plus, il est possible de proposer un choix parmi plusieurs fonctionnalités. On parle alors d'alternative.

L'exemple de la figure 2.14 est tiré de [KCH+90]. Il présente de manière très simplifiée les fonctionnalités d'une voiture. On peut constater que la climatisation est une fonctionnalité facultative, que le moteur et sa puissance sont obligatoires, et qu'il existe une alternative entre la boîte de vitesses manuelle et automatique. De plus, le diagramme présente une contrainte additionnelle qui établit que la climatisation nécessite un moteur présentant une puissance minimale. Dans le cadre général, il existe deux contraintes de base :

- **l'implication :** comme dans l'exemple précédent, la sélection d'une fonctionnalité implique la sélection d'une autre fonctionnalité (dans l'exemple ci-dessus, en restreignant la portée du choix) ;
- **l'exclusion :** dans ce cas, la sélection d'une fonctionnalité entraîne l'impossibilité de sélectionner une autre fonctionnalité.

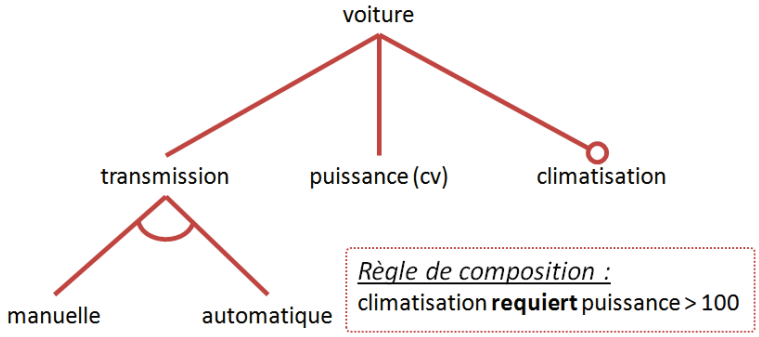

FIGURE 2.14 – Diagramme de fonctionnalités [KCH+90].

2.4.2.3 Extensions à FODA

A la suite de l'introduction des diagrammes de fonctionnalités, plusieurs évolutions ont été proposées. Certaines proposent des syntaxes différentes, d'autres un élargissement des concepts présents dans FODA. C'est ce second groupe que nous allons approfondir, à l'aide de la classification proposée dans [CHE05].

Initialement, les cardinalités avaient été écartées dans FODA. De nombreuses contributions ont proposé leur introduction [CHE04, CK05], parfois en s'inspirant d'UML® [GFd98, RBSP02, CPRS04]. Dans ces approches, les fonctionnalités optionnelles et obligatoires deviennent simplement des cas particuliers de cardinalités : [0..1] et [1..1] respectivement. En plus de proposer des cardinalités sur les fonctionnalités, l'introduction de cardinalités a aussi été proposée pour les alternatives. Dans FODA, il n'est possible de choisir qu'un seul élément dans un ensemble. Un concept de groupe a été introduit dans une extension afin de permettre plusieurs sélections dans ce même ensemble [RBSP02].

L'ajout d'attributs sur les fonctionnalités est proposé par [CBUE02]. De l'avis des auteurs, l'objectif est de simplifier certains diagrammes. En effet, il est possible de créer une fonctionnalité pour chaque valeur d'attribut (et éviter ainsi leur utilisation) mais cela conduit trop rapidement à de très grands diagrammes.

Plusieurs auteurs proposent des relations de différentes natures. La contribution la plus significative est celle de [LKL02] qui introduit trois nouveaux modes de relation. La relation *composé-de* est utilisée s'il existe une relation d'inclusion entre une fonctionnalité et une sous-fonctionnalité. Dans le cas où une fonctionnalité est une généralisation d'une sous-fonctionnalité, celles-ci entretiennent des relations *généralisation/spécialisation*. Enfin, la relation *implanté-par* est utilisée quand une fonctionnalité est utilisée pour en implanter une autre.

Quelques contributions se sont attachées à caractériser les fonctionnalités en le considérant sous différents aspects [CHE05]. Par exemple, dans l'approche RSEB [GFd98] un premier modèle est créé avec des fonctionnalités vues sous le prisme fonctionnel (comme dans FODA). Dans un second diagramme, les fonctionnalités sont vues sous l'angle architectural, qui caractérise le système en fonction de son architecture et non de ses fonctionnalités. Enfin, un troisième modèle présente les fonctionnalités sous le prisme de leur implantation.

Enfin, pour simplifier la lisibilité de grands digrammes de fonctionnalités, T. Bednasch [Bed02] propose de modulariser ceux-ci en plaçant certaines branches d'un diagramme de fonctionnalités dans des sous-diagrammes.

2.4.2.4 Modèle de variabilité orthogonale

Introduction

Le modèle de variabilité orthogonale (*Orthogonal Variability Model*, OVM) a été proposé par Klaus Pohl *et al.* dans [PBvdL05]. Pour définir l'objectif de ce modèle, commençons par une définition générale :

> "An orthogonal variability model is a model that defines the variability of a software product line. It relates the variability defined to other software development models such as feature models, use case models, design models, component models, and test models." [PBvdL05]

Un modèle de variabilité orthogonale est qualifié d'orthogonal car il modélise la variabilité à part, dans une autre dimension, séparée de l'implantation proprement dite des composants qui prennent part à cette variabilité. Il propose donc de poursuivre un double objectif :

- détailler et documenter la variabilité au sein d'une ligne de produits ;
- assurer un lien avec la documentation, afin d'aider la prise de décisions, de faciliter la communication et d'améliorer la traçabilité.

Nous allons à présent montrer comment ce double objectif est atteint, au travers de plusieurs concepts complémentaires et en relation, tirés de [PBvdL05].

Point de variation et variant

Les deux concepts de base du modèle de variabilité orthogonale sont le point de variation, et le variant. Un point de variation est un point de décision, qui permet d'effectuer un choix afin de personnaliser un produit au sein de la ligne de produit. Un variant est un morceau de produit, une fonctionnalité (visible ou non par l'utilisateur final) qui peut être ajouté à un produit, conformément aux points de variation.

Le modèle de variabilité orthogonale établit une distinction entre un point de variation interne et externe. Un point de variation interne a des variants associés qui ne sont visibles qu'aux développeurs et non à l'utilisateur final. Un point de variation externe, quant à lui, a des variants associés qui sont à la fois visibles au développeur et à l'utilisateur final. Cette distinction permet donc de savoir qui est concerné par le point de variation.

Entre les points de variation et les variants, il existe des liens qui permettent d'exprimer le fait qu'un point de variation offre certains variants. Ainsi, chaque point de variation doit être associé à au moins un variant et chaque variant à au moins un point de variation.

Alternative

Une alternative offre la possibilité de choisir un ou plusieurs variants au sein d'un groupe, sur la base de cardinalités :

> "The alternative choice groups a set of variants that are related through an optional variability dependency to the same variation point and defines the range for the amount of optional variants to be selected for this group." [PBvdL05]

D'un point de vue conceptuel, une alternative selon le modèle de variabilité orthogonal est donc très proche d'une alternative selon FODA. Il existe cependant une différence entre les deux : FODA ne définit pas de cardinalités dans les alternatives. Il faut pour cela recourir à une extension du langage.

Contraintes

Pour affiner la sélection des variants, six types de contraintes peuvent être exprimées.

La première est une contrainte d'implication : si un variant V1 est sélectionné, alors un variant V2 doit aussi être sélectionné. La seconde est une contrainte d'exclusion : si un variant V1 est sélectioné, alors un variant V2 ne peut pas être sélectionné. Notons que ces deux premières contraintes sont extrêmement classiques et qu'elles sont d'ailleurs présentes dans FODA.

Les contraintes suivantes sont quant à elles, moins habituelles. Les deux suivantes mélangent variants et points de variation. Ainsi, dans la troisième contrainte, si un variant V1 est sélectionné, alors le point de variation P1 doit être considéré et un choix doit être fait sur celui-ci. La quatrième contrainte procède de manière analogue, mais en excluant un point de variation au lieu de l'inclure.

Enfin, les deux dernières contraintes ne concernent que les points de variation. La cinquième est une contrainte d'implication : si un point de variation P1 est sélectionné, alors un point de variation P2 doit aussi être pris en considération. La sixième et dernière fait de même, avec l'exclusion.

Traçabilité entre le modèle de variabilité et les autres artéfacts de développement

Pour les concepteurs du modèle de variabilité orthogonale, la définition des points de variations, variants, alternatives et contraintes n'est qu'une part du travail de modélisation de la variabilité. En effet, il faut aussi établir un lien entre le modèle de variabilité et les autres modèles : analyse des besoins, documents de conception, code et tests.

Pour cela, le concept d'artéfact de développement est défini. Il permet d'abstraire n'importe quel document, formel ou non. Pour lier ces artéfacts de développement au modèle présenté précédemment, un lien nommé *représenté par* peut être ajouté entre un point de variation et un artéfact de développement. Les cardinalités sont définies de la manière suivante :

- un artéfact de développement peut être lié, ou non, à un nombre quelconque de points de variation ;
- un point de variation peut être lié, ou non, à un nombre quelconque d'artéfacts de développement.

2.4.2.5 FODA ou Modèle de variabilité orthogonale ?

Les concepts élaborés par FODA et le modèle de variabilité orthogonale sont extrêmement proches. Dans leur comparaison, certains auteurs ont beaucoup mis l'accent sur l'aspect centré sur les fonctionnalités le modèle de fonctionnalité, et centré sur les décisions pour le modèle de variabilité orthogonal [CGR+12]. Ainsi, FODA permet mieux d'exprimer ce qui est commun entre les produits de la ligne de produits, là où le modèle de variabilité orthogonale se focalise exclusivement sur les différences, au travers des ponts de décisions.

La différence la plus visible concerne la structure de représentation. Les diagrammes de fonctionnalités sont toujours représentés sous forme d'un arbre, avec une unique racine, alors que les modèles de variabilité orthogonales peuvent être représentés sous forme de forêt [RF09]. Derrière cette différence, c'est le processus d'élaboration des produits qui varie. Avec les diagrammes de fonctionnalités, celui-ci se fait de manière hiérarchique, en parcourant un arbre. Avec le modèle de variabilité orthogonale, l'approche est davantage linéaire, en explorant des listes ou des tables de décisions [CGR+12]. En fait, le modèle de variabilité orthogonale s'est largement appuyé sur FODA, qui le précédait de quinze ans. Il est ainsi venu combler certains manques, en particulier :

- l'ajout de cardinalités pour les alternatives ;
- une liste de contraintes plus élaborées ;
- un lien de traçabilité avec les artéfacts de développement ;
- une formalisation plus poussée, là où FODA avait centré ses efforts sur la notation graphique, même si ce travail a été effectué dans un second temps.

En soi, le modèle de variabilité orthogonale n'a donc pas apporté de nouvel élément majeur par rapport aux extensions proposées à FODA. Par contre, il a proposé d'intégrer de manière cohérente un certain nombre de propositions, les a organisées et formalisées. Cela justifie l'écho important de cette contribution dans la littérature [PBvdL05, CGR+12, RF09].

2.4.3 Ingénierie d'applications

Une fois l'ingénierie du domaine effectuée, la seconde étape nommée ingénierie d'applications a pour objectif la création des produits à proprement parler :

> "Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability." [PBvdL05]

Pour construire ces produits, on va s'appuyer sur les fonctionnalités (artéfacts de la définition précédente) et les contraintes identifiées lors de l'ingénierie du domaine. Pour créer un produit, on va prendre une série de décisions à chaque point de variabilité. On décidera, par exemple, de sélectionner ou non une fonctionnalité optionnelle ou d'une alternative dans un choix de type *un parmi n*. Une fois tous les choix effectués et donc que tous les points de variabilité ont été résolus, on obtient une configuration, qui liste l'ensemble des choix et qui aboutit à la création d'un produit.

Ce processus qui permet d'obtenir une configuration, peut être conduit en une seule fois ou à l'aide de raffinements successifs. Dans ce second cas, plusieurs intervenants ou groupes d'intervenants peuvent concourir à la création du produit. De plus, la configuration peut être décidée tôt dans le processus d'élaboration des produits ou, au contraire, retardée au maximum [CHE05].

Pour assister le travail de configuration, des outils ont vu le jour dans deux directions principales. Tout d'abord, certains se sont concentrés sur la vérification formelle des configurations. En effet, avec des règles en grand nombre et extrêmement complexes, la simple vérification de la validité d'une configuration nécessite soit un travail manuel complexe, soit une formalisation poussée. C'est cette dernière alternative qui a, par exemple, été utilisée par [Man02].

D'autres outils se sont concentrés sur le processus de création de produits. Ceux-ci sont qualifiés de configurateurs [AMS07] ; car ils permettent la création des configurations qui déterminent les produits. En fonction des choix successifs effectués par l'utilisateur, le configurateur va effectuer une série de déductions et ne présenter que les choix pertinents pour les points de variabilité suivants [BTRC05]. De cette manière, l'utilisateur évite des erreurs tout en gagnant du temps. A tout moment, l'utilisateur peut demander au configurateur de terminer une configuration. Le configurateur fera un ensemble de choix qui respectent les contraintes de la ligne de produits.

2.4.4 Bénéfices liés à l'utilisation de lignes de produits

Avec le recul dont nous disposons aujourd'hui, nous pouvons constater que l'approche proposée par les lignes de produits a apporté des gains très significatifs dans de nombreuses entreprises. Dans [Nor02], L. M. Northrop cite plusieurs exemples dont le premier est celui de CelsiusTech Systems. CelsiusTech Systems est une entreprise suédoise de défense qui fournit aux marines du monde entier des panneaux de commande et de contrôle. A l'aide d'une approche ligne de produit, ils ont livré plus de 50 systèmes sur la même base de composants réutilisables. Ils ont ainsi raccourci le délai de livraison de plusieurs années, réduit leur équipe en augmentant leur capacité de production, et atteint un niveau de réutilisation de logiciel de l'ordre de 90%.

Au-delà des réussites particulières, une liste de bénéfices liés à l'utilisation des lignes de produits a été établie par [PBvdL05]. Nous allons à présent lister les trois principaux. En cherchant à maximiser la réutilisation, les lignes de produits permettent des réductions importantes de coûts à partir d'un certain nombre de produits conçus.

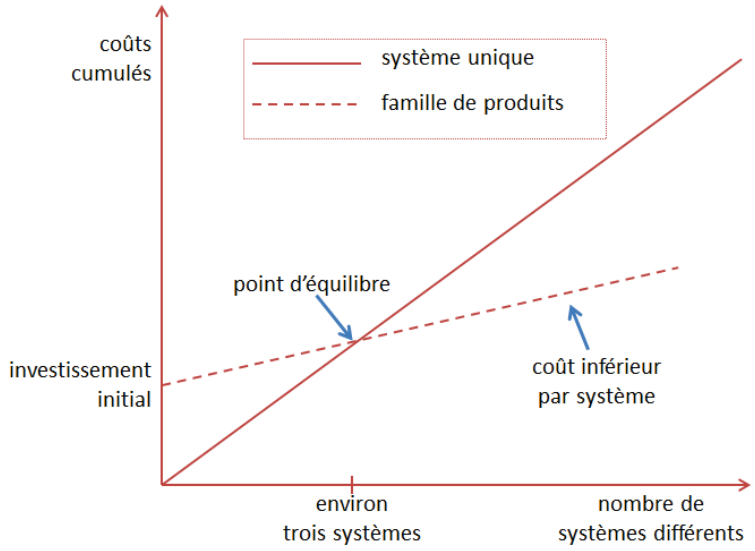

FIGURE 2.15 – Coûts de développement avec ou sans ligne de produits [PBvdL05].

La figure 2.15 est tirée de [PBvdL05]. Sur celle-ci, une ligne continue indique le coût cumulé du développement de systèmes différents en fonction de leur nombre, quand ceux-ci sont conçus individuellement. La courbe en pointillés fait de même, dans le cadre d'une ligne de produits. Pour de très faibles quantités, il est plus avantageux de ne pas recourir à une ligne de produits, en raison de son investissement initial incompressible. Le coût est identique pour trois systèmes et il est plus avantageux d'utiliser une ligne de produits au-delà. Selon [WL99] (cité par [PBvdL05]), ce point d'équilibre se situe entre trois et quatre systèmes différents.

Le second bénéfice est une amélioration de la qualité des produits. En effet, en étant intégrées dans plusieurs produits, les fonctionnalités sont utilisées dans des contextes différents et à plus large échelle. Ainsi, tout dysfonctionnement qui serait passé au travers de la phase de test, a plus de chances d'être repéré tôt et donc corrigé.

Enfin, l'utilisation de ligne de produits permet de réduire le délai de mise sur le marché (*time to market*). Plus précisément, si la création de la ligne de produits prend du temps et retarde la mise sur le marché des premiers produits, les suivants peuvent être conçus extrêmement rapidement.

2.4.5 Synthèse

Les lignes de produits ont montré de longue date leur capacité à prendre en charge la production d'un ensemble de produits différents d'une même famille, en réduisant de manière extrêmement importante les coûts et les délais de fabrication. Pour cela, celles-ci sont organisées autour de deux processus : l'ingénierie du domaine et l'ingénierie applicative. Dans ces deux activités, le concept de variabilité est absolument central :

> "The explicit definition and management of variability distinguishes software product line engineering from both single-system development and software reuse." [PBvdL05]

Cette gestion de la variabilité, qui permet d'exprimer un ensemble de contraintes au niveau architectural répond à la préoccupation de contrôle que nous avons soulevée à l'issue de la présentation de l'approche à services. Cependant, les lignes de produits ne permettent pas à proprement parler l'adaptabilité. Elles visent la conception d'un ensemble de produits différents au sein d'une même famille et non la modification d'un produit existant.

Pour prendre en compte cette problématique d'adaptation tout en conservant les bénéfices de la gestion de la variabilité au niveau architectural, une nouvelle approche a été proposée : les lignes de produits dynamiques. Dans ce cadre, l'objectif n'est plus de permettre la production d'un ensemble de produits d'une même famille, mais d'un unique produit, pouvant être adapté de manière extrêmement contrôlée, afin de le faire évoluer d'une architecture à une autre.

Nous allons à présent présenter cette approche, ainsi qu'un ensemble de travaux permettant d'appréhender la richesse des propositions effectuées.

2.5 Lignes de produits dynamiques

2.5.1 Introduction

Suite aux succès enregistrés par les lignes de produits logicielles, une autre approche, nommée ligne de produits dynamiques a vu le jour. Avant de décrire celle-ci plus en détails, il est important de noter que ce champ de recherche est encore en train de se structurer. En effet, contrairement aux lignes de produits traditionnelles qui disposent désormais de concepts qui font l'objet de consensus larges, les lignes de produits dynamiques ne disposent pas aujourd'hui de ce consensus, et au-delà de celui-ci, de la même maturité [HPS12]. Nous allons donc détailler ici les principes les plus largement acceptés, qui permettent de comprendre l'approche proposée par lignes de produits dynamiques [HHPS08, HHPS13].

La rupture centrale apportée par les lignes de produits dynamiques concerne le moment auquel les décisions relatives aux points de variations sont prises. Avec les lignes de produits traditionnelles, toutes les décisions sont prises lors de la conception. Les lignes de produits dynamiques adoptent une attitude différente sur ce point.

> "DSPLs bind variation points at runtime, initially when software is launched to adapt to the current environment, as well as during operation to adapt to changes in the environment." [HHPS08]

La citation précédente indique que des décisions concernant les points de variation sont prises lors de l'exécution et non plus avant celle-ci. Les lignes de produits dynamiques proposent donc de configurer ou de reconfigurer un logiciel lors de son exécution, en s'appuyant sur des points de variation bien connus dans le cadre des lignes de produits traditionnelles.

> "A software reconfiguration pattern is a solution to a problem in a software product family where the configuration needs to be updated while the system is operational. It defines how a set of components participating in a software pattern cooperate to change the configuration of a system from one configuration of the product family to another." [GH04]

Pour cela, un patron de reconfiguration décrit la nouvelle configuration qui doit être appliquée au système, sans son arrêt. On passe ainsi d'une configuration valide à une autre lors de l'exécution. La ligne de produits n'est donc plus utilisée pour créer des produits différents, mais pour permettre à un même produit de passer d'un état à un autre, potentiellement de nombreuses fois au cours de son exécution [HPS12]. En procédant de cette manière, [MBJ+09] compare le système à une machine à états, où les états sont les configurations possibles du système et les transitions les chemins de migration.

Nous pouvons donc noter que la gestion de la variabilité, centrale dans le cadre des lignes de produits traditionnelles se retrouve aussi au premier plan dans le cadre des lignes de produits dynamiques. Elle fait d'ailleurs l'objet de nombreux travaux de recherche, en

particulier, en ce qui concerne la modification de la politique de variabilité à l'exécution [CB11]. Dans ce cadre, le concept de méta-variabilité a émergé [HWS+09]. Il propose de faire évoluer la variabilité à l'exécution, autrement dit, de modifier la politique de variabilité de la ligne de produits dynamique lors de son exécution. Avec cette approche, il est donc possible d'ajouter ou de supprimer des variants alors que le système s'exécute [HWS+09]. Cela conduit à effectuer une distinction entre la variabilité ouverte et la variabilité fermée. La variabilité ouverte se réfère à la méta-variabilité, alors que la variabilité fermée concerne la variabilité qui ne peut plus être modifiée une fois le système exécuté.

Au-delà de la variabilité ouverte ou fermée, si tous les points de variabilité peuvent être décidés lors de l'exécution, des approches mixtes peuvent être envisagées. Celles-ci permettent de prendre certains choix avant l'exécution, afin de préconfigurer le système en fonction de propriétés statiques de l'environnement, tout en laissant ouverts d'autres choix pour l'exécution.

Contrairement à d'autres approches qui cherchent à automatiser le processus d'adaptation (comme l'informatique autonomique) les lignes de produits dynamiques ne font aucune hypothèse sur la nature automatisée de ce processus [HHPS13]. Le passage d'une configuration peut donc aussi bien être assuré par une intervention humaine outillée, que par une entité logicielle autonome. Cela dit, de très nombreux travaux actuels visent une forme d'automatisation plus ou moins poussée, comme nous allons le voir dans les approches décrites au paragraphe suivant.

2.5.2 Critères de caractérisation

Pour caractériser les approches des lignes de produits dynamiques que nous allons présenter, nous avons choisi de nous attacher aux critères suivants.

Le premier concerne la perception du contexte d'exécution. Les lignes de produits dynamiques ont pour objectif de s'adapter de manière contrôlée à un environnement d'exécution mouvant. Aussi la perception de cet environnement est une tâche d'une grande importance. Ce premier critère va donc s'attacher à définir la technologie utilisée par chaque approche. Celle-ci peut être un *framework* existant qui est alors réutilisé tel quel ou adapté. Ce peut aussi être une implantation spécifique à la ligne de produits considérée.

Les lignes de produits dynamiques héritent des lignes de produits traditionnelles une gestion stricte de la variabilité, comprise alors comme l'ensemble des configurations possibles pour une application. Un second critère s'attachera à définir la manière avec laquelle cette variabilité est définie. Certaines lignes de produits dynamiques ont cherché à se placer dans l'héritage des lignes de produits traditionnelles, en utilisant un diagramme de fonctionnalités ou un modèle de variabilité orthogonale. Ceux-ci ont été largement étudiés et ils disposent de plus d'un outillage très important qui peut être réutilisé. D'autres approches ont privilégié des modèles spécifiques.

Le troisième critère retenu se concentrera quant à lui sur le procédé de calcul de l'architecture cible que l'on désire atteindre. Pour cela, il existe deux grandes familles d'approches [BHSdA12]. La première est basée sur des tuples événement – condition – action (ECA).

Avec cette approche, lors d'un événement, une série de conditions sont évaluées. Si celles-ci sont satisfaites, les actions correspondantes sont exécutées. Cette approche offre la possibilité d'une validation des règles lors de la conception et se révèle extrêmement rapide à l'exécution. Cependant, elle nécessite de lister de manière exhaustive tous les événements et les plans de reconfiguration possibles, ce qui se révèle dans les faits très difficile quand le nombre d'événements augmente. La seconde approche est basée sur l'utilisation d'une fonction qui va évaluer la performance d'un plan de reconfiguration à prendre en charge une situation donnée. Dans ce cadre, chaque solution potentielle est évaluée et celle obtenant le meilleur score est choisie. Cette approche ne nécessite donc pas l'énumération de tous les plans de reconfiguration à la conception, mais implique en contrepartie un temps de calcul à l'exécution qui peut devenir très long dès que le nombre d'alternatives de reconfiguration augmente.

Une fois l'architecture cible calculée, il est nécessaire de reconfigurer l'application. Un quatrième critère va s'attacher à qualifier la méthode employée pour cela. Les modèles permettent de prendre en charge cette préoccupation avec un haut niveau d'abstraction et une synchronisation avec la plate-forme sous-jacente si ceux-ci sont causaux. Une autre méthode utilise la reconfiguration dynamique d'une architecture à base de composants, par mise à jour des liens entre ceux-ci et parfois, instanciation et destruction de composants. Enfin, une troisième possibilité est l'utilisation de la programmation orientée aspects [FECA04]. Avec cette approche, une série de préoccupations est implantée dans du code qui peut être tissé ou séparé de l'application à la volée, mettant ainsi à jour l'application globale.

Quelle que soit le processus utilisé pour mettre en œuvre l'application, son adaptation dynamique nécessite une plate-forme dynamique, qui permette l'adaptation à chaud des applications. Celle-ci est souvent masquée par des mécanismes de plus haut niveau, mais elle demeure néanmoins la condition de possibilité de toutes les adaptations. Aussi, nous indiquerons cette plate-forme dans un cinquième critère

Enfin, un sixième critère donnera le champ d'application des lignes de produits que nous allons détailler. Certaines cherchent à être extrêmement généralistes, d'autres visent un domaine particulier, comme par exemple la maison numérique.

2.5.3 Différents travaux

Nous allons à présent détailler les principaux travaux, en axant la description sur leur originalité propre et la prise en compte des dimensions que nous venons de citer.

2.5.3.1 SESAMO

SESAMO [CFP08, CGFP09a] propose une approche pour bâtir des applications dans le domaine des systèmes pervasifs et, en particulier, de la maison numérique (*smart homes*). Pour cela, C. Cetina s'est inspirée des travaux sur la sélection automatisée de variants. Dans SESAMO, les modèles de fonctionnalités sont transférés à l'exécution, pour permettre la reconfiguration dynamique lors d'une modification du contexte d'exécution perçue par le

framework PervML [MP06]. Lorsque cela est nécessaire, une nouvelle architecture cible est calculée et un plan de reconfiguration exécutable est défini avant d'être appliqué [CGFP09b].

Cette approche propose donc d'effectuer des reconfigurations sur la base des fonctionnalités. Pour cela, SESAMO utilise le modèle de fonctionnalités FAMA [BTRC05] qui se prête bien aux raisonnements automatisés. Notons que l'utilisation de CVL (*Common Variability Language*) a fait l'objet d'une étude spécifique dans [CHZ+09].

L'approche proposée cherche à couvrir l'ensemble du cycle de développement, depuis la conception jusqu'à l'exécution. En particulier, il est possible, lors de la conception, de vérifier la validité des reconfigurations qui pourront être appliquées lors de l'exécution. Une implantation a été effectuée directement au-dessus de la plate-forme OSGi™.

Les caractéristiques clé de l'approche SESAMO sont résumées dans le tableau 2.1.

Perception du contexte	Réutilisation du *framework* PervML [MP06]
Expression de la variabilité	Modèle de fonctionnalités FAMA [BTRC05]
Calcul de l'architecture cible	ECA (*Trigger* dans PervML) [CFP08]
Mécanisme de reconfiguration	Dirigé par les modèles (MORE *Model-Based Reconfiguration Engine*) avec activation et désactivation de fonctionnalités
Plate-forme d'exécution	OSGi™ (et EMF)
Domaine principal d'application	*smart homes*

Tableau 2.1 – Résumé de l'approche SESAMO .

2.5.3.2 MADAM

L'approche proposée par MADAM (*Mobility and ADaptation-enAbling Middleware*) [HSSF06, GBE+09] repose sur l'utilisation de modèles architecturaux [FHS+06], basés sur un modèle à composants.

La principale source de variabilité provient de la substitution de composants au sein de l'architecture exécutée. Pour cela, plusieurs implantations peuvent être chargées pour un même composant. Pour choisir l'implantation à utiliser, les programmeurs peuvent placer lors de la conception des annotations sur les composants afin de décrire leur qualité de service. A l'exécution, l'adaptation est effectuée sur la base de ces propriétés à l'aide d'une fonction d'utilité qui classe les implantations en fonction de leur capacité à prendre en charge le contexte d'exécution courant. Cette fonction d'utilité traduit donc le but de l'application en un algorithme de classement. Une fois sélectionnée une implantation qui doit être modifiée, la plate-forme d'exécution prend en charge le processus de substitution.

Les caractéristiques clé de l'approche MADAM sont résumées dans le tableau 2.2.

Perception du contexte	Un *Context Manager* recense les valeurs de variables environnementales
Expression de la variabilité	Dans un modèle d'adaptation, obtenu après transformation d'un modèle UML® 2.0 en code Java™
Calcul de l'architecture cible	Sélection du meilleur variant par l'*Adaptation Manager* à l'aide d'une fonction d'utilité. L'évaluation est déclenchée par une notification du *Context Manager*
Mécanisme de reconfiguration	Sélection du meilleur variant par l'*Adaptation Manager* à l'aide d'une fonction d'utilité. L'évaluation est déclenchée par une notification du *Context Manager*
Plate-forme d'exécution	Approche indépendante de la plate-forme d'exécution (utilisation de MDA), mais implantation en Java™ (versions embarquées, en particulier J2EE CDC Personal edition)
Domaine principal d'application	Terminaux mobiles (en particulier, assistants numériques personnels)

Tableau 2.2 – Résumé de l'approche MADAM .

2.5.3.3 Trinidad *et al.*

Trinidad *et al.* [TCPB07] ont proposé une approche dans laquelle une architecture à composants est générée à partir d'un modèle de fonctionnalités. Pour assurer le passage de l'un à l'autre, une fonctionnalité ne peut être implantée que par un unique composant. L'architecture générée est capable d'activer ou de désactiver des composants à l'exécution, à la demande d'un configurateur qui effectue des opérations d'analyse sur le modèle de fonctionnalité afin de prendre des décisions [BSRC10].

Bien que conceptuellement intéressante, cette approche a fait l'objet de peu de publications. En effet, les auteurs ont préféré axer leurs travaux de recherche sur le *framework* FAMA [BSTRC07], qui effectue des raisonnements sur les diagrammes de fonctionnalités.

Les caractéristiques clé de l'approche de Trinidad *et al.* sont résumées dans le tableau 2.3.

Perception du contexte	Non couverte par l'approche
Expression de la variabilité	Modèle de fonctionnalités FAMA [BSTRC07]
Calcul de l'architecture cible	Par un configurateur, qui doit être implanté pour chaque application
Mécanisme de reconfiguration	Au niveau composant, un composant implantant une fonctionnalité
Plate-forme d'exécution	Java™ (mais possibilité avec tous les langages permettant le chargement dynamique de code [LSR07])
Domaine principal d'application	Pas de domaine particulier visé.

Tableau 2.3 – Résumé de l'approche de Trinidad *et al.* .

2.5.3.4 Genie (Bencomo *et al.*)

Genie [BGF+08, BBFS08] est un outil basé sur MetaEdit+ [TR03] qui prépare l'adaptation du système lors de sa conception. Pour cela, un DSL basé sur OVM permet de modéliser les différentes architectures valides qui pourront être exécutées par un modèle à composants lors de l'exécution. De plus, ce DSL permet de spécifier un ensemble de transitions, pour passer d'une architecture à une autre.

On obtient donc ainsi une machine à états qui permet de spécifier la logique d'adaptation du système. Sur les transitions, des conditions sont ajoutées pour décrire ce qui déclenchera à l'exécution le passage d'une architecture à une autre. De plus, des scripts sont préparés pour pouvoir assurer les opérations de reconfiguration sur la plate-forme lors de l'exécution. Pour alléger le travail du développeur, les squelettes des scripts sont générés pour chaque transition à partir du DSL. L'exécution est basée sur la plate-forme Gridkit [GCBP05], qui permet de réifier l'architecture exécutée par le modèle à composants dynamiques sous-jacent.

Le principal problème de cette approche est la quantité de scripts de reconfiguration que le développeur doit écrire. Quand le nombre d'états du système augmente, le nombre de transitions et donc de scripts de reconfiguration peut potentiellement augmenter selon le carré du nombre des états.

Les caractéristiques clé de l'approche Genie sont résumées dans le tableau 2.4.

Perception du contexte	Middleware permettant la découverte dynamique de services [FCBG07]
Expression de la variabilité	DSL basé sur OVM
Calcul de l'architecture cible	Politiques d'adaptation selon l'approche ECA
Mécanisme de reconfiguration	Au niveau composant
Plate-forme d'exécution	Gridkit [GCBP05]
Domaine principal d'application	Grilles (*grid computing*), informatique mobile, systèmes embarqués

Tableau 2.4 – Résumé de l'approche Genie.

2.5.3.5 CAPucine

CAPucine (*Context-Aware Service-Oriented Product Line*) [PBD09, Par11, PBCD11, PQD12] est une chaîne de production logicielle, qui propose la conception et l'exécution de logiciels adaptatifs dans une approche ligne de produits dynamiques.

Pour cela, un méta-modèle générique permet de définir les différentes fonctionnalités sous forme de modèles d'aspects. Pour concevoir le produit initial, la sélection des fonctionnalités entraîne la combinaison des modèles d'aspects correspondants en un unique modèle. Afin de produire le code source, une approche suivant le paradigme de l'ingénierie dirigée par les modèles est employée. Le modèle obtenu est transformé en prenant à la fois en

compte les caractéristiques de la plate-forme d'exécution (FraSCAti, une plate-forme à composants SCA) et celles du langage de programmation cible (Java™). On obtient alors le code source de notre application.

L'adaptation dynamique repose sur le méta-modèle générique utilisé lors de la phase de conception et sur un processus de tissage dynamique d'aspects. Pour cela, les modifications de contexte sont détectées par le *framework* COSMOS [RCS08]. Un changement de contexte entraîne la sélection d'un nouvel ensemble de variants. Cette configuration est alors validée, puis la différence entre celle-ci et la configuration actuelle est calculée. De cette différence, des scripts de reconfiguration sont appliqués. Ceux-ci prennent en charge le tissage de nouveaux aspects et le retrait de ceux qui doivent être supprimés.

Les caractéristiques clé de l'approche CAPucine sont résumées dans le tableau 2.5.

Perception du contexte	*Framework* COSMOS [RCS08]
Expression de la variabilité	Méta-modèle spécifique créé avec EMF qui met en avant les points de variation et les variants
Calcul de l'architecture cible	Par un *adapter* qui définit une configuration cible en termes de variants lors d'une modification de contexte d'exécution
Mécanisme de reconfiguration	Tissage dynamique d'aspects, piloté par des scripts de reconfiguration
Plate-forme d'exécution	FraSCAti
Domaine principal d'application	Pas de domaine particulier visé.

Tableau 2.5 – Résumé de l'approche CAPucine.

2.5.3.6 DiVA

L'approche proposée dans le cadre du projet DiVA [MBNJ09, MBJ+09] se place dans la continuité de CAPucine, précédemment décrite. Elle combine l'utilisation d'aspects et d'ingénierie dirigée par les modèles pour :

- luter contre l'explosion combinatoire dans l'énumération du nombre de configurations possibles lors de l'exécution, en se centrant sur les variants, au lieu des configurations ;
- éviter d'avoir à écrire à la main chaque script de reconfiguration ;
- prendre en compte les spécificités associées aux protocoles de communication, sans pour autant faire augmenter le nombre d'états possibles du système comme dans l'approche Genie.

Des aspects sont associés aux variants, selon l'approche des *SmartAdapters* définie dans [LMV+07, MBJ08]. Lors de l'exécution, une nouvelle sélection de variants entraîne ainsi une nouvelle sélection d'aspects. Cette nouvelle configuration peut alors être validée, puis la différence entre l'architecture actuelle et l'architecture souhaitée est calculée. A l'aide d'une

approche dirigée par les modèles, un script de reconfiguration est calculé pour la plate-forme d'exécution, afin de mettre à jour le tissage des aspects dans le bon ordre. L'ensemble de ces étapes repose sur l'utilisation d'un méta-modèle d'adaptation [MFB+08], contenant quatre modèles en relation :

- un modèle de variabilité, donne l'ensemble des variants, sous la forme d'une liste plate ou d'un diagramme de fonctionnalités ;
- un modèle de dépendances définit des propriétés invariantes et des contraintes qui permettent la validation d'un ensemble de règles avant l'exécution avec des techniques de vérification de modèle [FDB+09] ;
- un modèle du contexte d'exécution permet de percevoir l'environnement, pour réagir aux modifications de celui-ci ;
- un modèle de logique d'adaptation, modélise les modifications à apporter au système (en termes de variants), en fonction de l'environnement.

Enfin, l'exécution est basée sur une plate-forme disposant d'un modèle causal, afin de faciliter l'application des modifications grâce à une montée en abstraction.

Les caractéristiques clé de l'approche DiVA sont résumées dans le tableau 2.6.

Perception du contexte	EnTiMid [NDBJ08]
Expression de la variabilité	Modèle de variabilité et modèle de dépendances
Calcul de l'architecture cible	Les aspects liées à des éléments du contexte sont sélectionnés quand cela devient nécessaire [FDB+09], les autres peuvent être activés à la demande de l'utilisateur
Mécanisme de reconfiguration	Génération de scripts qui modifient le tissage d'un ensemble d'aspects
Plate-forme d'exécution	Approche indépendante d'une plate-forme spécifique, mais possibilité d'utilisation de Fractal, OpenCom...
Domaine principal d'application	Pas de domaine particulier visé.

Tableau 2.6 – Résumé de l'approche DiVA .

2.5.3.7 MUSIC

MUSIC [REF+08, FHS08, RBD+09, JHB+10] est un projet basé sur les résultats de MADAM, afin d'étendre les mécanismes obtenus à l'approche orientée services. Pour cela, la plate-forme à composants basée sur OSGi™ offre la possibilité de modifier dynamiquement les liaisons vers des services, s'adaptant ainsi à leur disponibilité. Elle permet aussi de prendre en compte leur qualité de service afin d'effectuer la liaison la plus adaptée à un contexte donné. Pour cela, MUSIC définit un *framework* d'adaptation chargé d'évaluer chaque alternative, afin de sélectionner celle qui apportera la meilleure qualité de service à l'application.

Les caractéristiques clé de l'approche MUSIC sont résumées dans le tableau 2.7.

Perception du contexte	Un *Context Manager* recense les valeurs de variables environnementales
Expression de la variabilité	Dans un *plan repository*, mis à jour par le *Discovery Service* qui ajoute ou retire des *plan variants* en fonction de l'arrivée ou du départ de services
Calcul de l'architecture cible	Sélection du meilleur variant par un *adaptation planning process* à l'aide d'une fonction d'utilité. L'évaluation est déclenchée par une notification du *Context Manager*
Mécanisme de reconfiguration	Au niveau composant et service
Plate-forme d'exécution	OSGi™
Domaine principal d'application	Informatique ubiquitaire

Tableau 2.7 – Résumé de l'approche MUSIC .

2.5.3.8 SASSY project

L'approche proposée par SASSY [MEM+09, GH11, MGMS11] (*Self-Architecting Software Systems*) propose de prendre en compte le primodéploiement d'une application, qui sera adaptable par la suite. Le primo-déploiement est traité comme une activité de ligne de produits classique, avec l'ingénierie du domaine et l'ingénierie applicative. La reconfiguration du système est perçue comme une tâche spécifique.

A l'exécution, l'application est structurée en trois couches, selon l'approche proposée par [KKP+09]. Au plus haut niveau se trouve le raisonnement sur le diagramme de fonctionnalités de PLUS [Gom04], qui permet de déterminer l'architecture la plus adaptée à chaque instant. Cette brique de décision repose sur une fonction d'utilité paramétrable avec des buts de qualité de service fournis par l'administrateur [MEG+10]. Au plus bas niveau se trouve une plate-forme à composants. Entre les deux, une couche intermédiaire fournit à la couche haute une vision de la plate-forme d'exécution et pilote la modification de celle-ci en transformant une liste de fonctionnalités en un ensemble d'ordres de reconfiguration (en utilisant la table de correspondance fonctionnalité/composant de PLUS). La reconfiguration de l'architecture à composants se fait selon des patrons expliqués dans [GH12].

Les caractéristiques clé de l'approche du projet SASSY sont résumées dans le tableau 2.8.

Perception du contexte	*SASSY monitoring service*
Expression de la variabilité	Diagramme de fonctionnalités de PLUS [Gom04]
Calcul de l'architecture cible	Fonction d'utilité, qui prend en compte la qualité de service
Mécanisme de reconfiguration	Au niveau composant et service
Plate-forme d'exécution	Eclipse Swordfish (basé sur OSGi™) et Apache CXF
Domaine principal d'application	Pas de domaine particulier visé.

Tableau 2.8 – Résumé de l'approche SASSY .

2.5.3.9 Approche de Lee *et al.*

L'approche de Lee *et al.* [LMN08, KLR09, LKR12, LK13] est basée sur une méthodologie d'analyse de fonctionnalités pour identifier les services qui devront être implantés. Une originalité de ces travaux est de prévoir une implantation soit sous forme de workflow d'orchestration de services, soit sous la forme de services atomiques. Une fois ces éléments définis, ils sont liés à la définition d'un contexte utilisateur et des contraintes additionnelles en termes d'invariants peuvent être exprimées.

A l'exécution, la plate-forme interagit avec des fournisseurs de services pour sélectionner ceux qui présentent la qualité de service la mieux adaptée. Cette sélection est faite à l'aide d'un fichier de description XML qui permet d'exprimer des contraintes sur les propriétés non-fonctionnelles des services. Un sous-système surveille la qualité de service demandée par l'utilisateur et adapte en conséquence la composition dans le cas où celle-ci ne soit pas respectée.

Les caractéristiques clé de l'approche de Lee *et al.* sont résumées dans le tableau 2.9.

Perception du contexte	Par un *Context Analyzer* qui donne la liste des services disponibles
Expression de la variabilité	Modèle de fonctionnalités étendu avec les notions de service et de workflow d'orchestration
Calcul de l'architecture cible	Fonction d'utilité, qui permet de maximiser la qualité de service obtenue après négociation avec les différents services disponibles
Mécanisme de reconfiguration	Niveau composant
Plate-forme d'exécution	Apache River
Domaine principal d'application	Pas de domaine particulier visé.

Tableau 2.9 – Résumé de l'approche Lee et al .

2.5.4 Synthèse des lignes de produits dynamiques

Les lignes de produits dynamiques cherchent à la fois à concilier :

- une gestion stricte de la variabilité, issue des lignes de produits : grâce à celle-ci, un contrôle extrêmement fin de l'architecture est possible ;
- la possibilité d'adapter ou d'auto-adapter les systèmes lors de leur exécution.

La fusion de ces deux préoccupations nous semble extrêmement pertinente. Elle permet de dépasser la limite évoquée à l'issue de la partie sur l'approche à services, en apportant le cadre issu des lignes de produits afin de guider et valider les modifications architecturales.

2.6 Conclusion

La problématique d'adaptation prend de plus en plus d'importance dans l'administration des applications modernes. L'évolution fréquente des besoins, les changements de disponibilité des ressources, la nécessité de corriger des dysfonctionnements et la prise en compte de la mobilité des utilisateurs [Gar13] nécessitent des interventions complexes et fréquentes sur les éléments logiciels.

La mise en œuvre de ces adaptations peut être effectuée à plusieurs niveaux d'abstraction. Au niveau code, au niveau objet, au niveau composant et au niveau architecture. Nous avons vu que le niveau architectural est le plus adapté afin de mener à bien le processus d'adaptation. En effet, lui seul permet de manipuler des applications de grande taille tout en conservant une vue d'ensemble et en formalisant des contraintes de conception afin de s'assurer que les modifications proposées restent valides.

Malheureusement, si l'adaptation au niveau architectural est perçue depuis de nombreuses années comme souhaitable [OGT+99], elle n'a pas encore atteint aujourd'hui un degré de maturité suffisant [GAWM11]. De nombreux travaux sont encore en cours, afin d'identifier les patrons de conception qui permettront de structurer de la manière la plus efficace possible le processus d'adaptation au niveau architectural.

Parmi ceux-ci, nous nous sommes intéressés plus particulièrement aux architectures orientées services, qui permettent de bâtir des applications de grande taille avec le patron d'interactions à trois acteurs de l'approche orientée service. Cette approche offre un point de vue adapté pour comprendre l'application et l'adapter le cas échéant. En effet, le couplage large entre les consommateurs et les fournisseurs de services permet un découpage naturel de l'application. De plus, la nature même de l'approche orientée service permet de mettre en œuvre de la flexibilité au niveau de la structure architecturale. Si le dynamisme est pris en charge, il devient aussi possible d'adapter « à chaud » et de manière automatisée une application en changeant de fournisseur de services en s'adaptant au plus tôt à l'apparition ou à la disparition de services.

Cependant, si le paradigme de l'approche orientée service permet une grande souplesse on peut désirer a contrario mieux maîtriser les architectures produites, en définissant en

amont des contraintes supplémentaires pour guider la composition des applications. Cette préoccupation, en dehors du périmètre de l'approche orientée service, est centrale dans l'ingénierie des lignes de produits logicielles qui proposent de modéliser de manière extrêmement fine la variabilité lors de la conception. Les travaux issus des lignes de produits ont permis dans un second temps l'émergence des lignes de produits dynamiques, qui utilisent la modélisation de la variabilité lors de leur exécution, pour adapter ou auto-adapter les systèmes.

Cependant, malgré le cadre conceptuel offert par les lignes de produits dynamiques, l'automatisation du processus d'adaptation demeure extrêmement complexe. Aussi, de nombreux travaux se sont penchés sur cette tâche afin de fournir un patron de conception identifiant et séparant un ensemble de préoccupations d'adaptation afin de mieux traiter cette problématique.

Parmi les approches proposées, l'une d'entre elles a connu un succès important, et a d'ailleurs été utilisée dans le cadre des lignes de produits dynamiques : l'informatique autonomique. Le chapitre qui va suivre va donc présenter cette approche.

Chapitre 3

Informatique autonomique

Sommaire

3.1 Présentation de l'informatique autonomique 61
3.1.1 Origine de l'informatique autonomique 61
3.1.2 Définitions de l'informatique autonomique 62
3.1.3 Niveaux de maturité d'un système autonomique 63
3.1.4 Synthèse 65
3.2 Sources d'inspiration 66
3.2.1 La biologie 66
3.2.2 La théorie du contrôle 67
3.2.3 Autres sources d'inspiration 68
3.2.4 Synthèse 69
3.3 Les propriétés d'un système autonomique 70
3.3.1 Propriétés de base 70
3.3.2 Arbre de R. Sterritt et D. Bustard 70
3.3.3 Synthèse 72
3.4 Architecture des systèmes autonomiques selon IBM 73
3.4.1 Architecture des systèmes autonomiques 73
3.4.2 Architecture des éléments autonomiques 73
3.4.3 Architecture des gestionnaires autonomiques 76
3.4.4 Synthèse 77
3.5 Architectures pour représenter la connaissance 79
3.5.1 Introduction et critères de caractérisation 79
3.5.2 Le *framework* Rainbow 80
3.5.3 Plastik 85
3.5.4 Approche de J. Kramer et J. Magee 87
3.5.5 Prism-MW et DIF 88

3.5.6 Architectural Runtime Configuration Management (ARCM) . . 91
3.5.7 Synthèse 93
3.6 Conclusion 94

L'implantation de systèmes auto-adaptables repose sur une capacité de modification de la structure ou du comportement de ceux-ci. Cette capacité peut être obtenue à l'aide d'un ensemble de mécanismes d'adaptation que nous avons présenté dans le chapitre précédent. Si cette capacité d'adaptation est indispensable, elle nécessite aussi d'être guidée par des politiques de haut niveau qui traduisent les objectifs d'administration souhaités. De plus, une infrastructure est aussi nécessaire afin de percevoir l'état du système à administrer, de le comprendre, de définir et d'exécuter des actions correctives si cela est requis pour atteindre les objectifs demandés. L'informatique autonomique, proposée par IBM [Hor01], est un domaine d'étude qui recherche la meilleure façon d'implanter ces systèmes auto-adaptables.

Dans ce chapitre, nous allons présenter l'informatique autonomique, en commençant par la positionner par rapport à ses sources d'inspiration et des propriétés d'administration qu'elle désire garantir. Ensuite, nous détaillerons l'architecture MAPE-K proposée par IBM pour servir de guide à implantation des systèmes autonomiques. Enfin nous donnerons différents exemples d'implantation que nous analyserons, en portant notre attention sur la connaissance architecturale qui nous apparaît centrale afin d'administrer les systèmes complexes.

3.1 Présentation de l'informatique autonomique

3.1.1 Origine de l'informatique autonomique

L'informatique autonomique a été introduite par P. Horn, au cours d'un discours à l'université de Harvard en 2001. A la suite de celui-ci, il a publié un manifeste qui marque le début de cette approche [Hor01]. Il fait le constat que face à la complexité croissante des systèmes, même les personnes les plus expertes atteignent leur limite de compétence pour les installer, les configurer, les optimiser et les réparer dans un contexte où les corrections doivent être de plus en plus rapides. De plus, il apparaît que la seule augmentation du nombre d'administrateurs ne permettra pas de prendre en charge sur le long terme des applications dont la taille ne cesse de croître. De ce point de vue, la situation est analogue à ce qui s'est passé concernant la téléphonie. A l'origine, des opérateurs établissaient manuellement la mise en relation des correspondants, en branchant à la main des câbles pour établir une connexion de bout en bout. L'augmentation du nombre d'abonnés à rendu indispensable l'automatisation de ce processus de mise en relation [Mai02]. En effet, des analystes d'AT&T/Bell estiment que sans celle-ci, un million d'opérateurs aurait été nécessaires dans les années 1970 pour le seul territoire des Etats-Unis !

P. Horn propose comme réponse l'informatique autonomique, dans laquelle le système est autogéré, surveillé et adapté en permanence afin de garantir un fonctionnement optimal, masquant ainsi sa complexité aux administrateurs tout en les libérant des opérations répétitives de maintenance. Dans son manifeste, il dresse une liste de huit caractéristiques qui doivent être prises en compte par tout système autonomique :

- **le système doit se connaître lui-même :** il doit disposer d'une connaissance détaillée de ses composants, de son état courant, de ses capacités et des relations qu'il entretient avec d'autres systèmes. En effet, un système ne peut pas administrer quelque chose qu'il ne connaît pas, ou contrôler des points spécifiques si ceux-ci ne sont pas définis.
- **le système doit pouvoir se configurer et se reconfigurer lui-même dans un environnement mouvant et imprévisible :** le nombre de variables à ajuster pouvant se compter en centaines tout en nécessitant des délais de réaction très courts (parfois en secondes), les reconfigurations du système doivent être effectuées de manière automatisée ;
- **le système doit chercher en permanence un moyen d'optimiser son fonctionnement :** cet effort permanent est la seule solution pour gérer des demandes contradictoires et en constante évolution. Pour cela, le système doit disposer de boucles de rétroaction, inspirées de la théorie du contrôle ;
- **le système doit pouvoir se réparer :** il doit découvrir les dysfonctionnements ainsi que leurs causes et trouver, le cas échéant, un moyen alternatif pour utiliser les ressources ou reconfigurer le système ;
- **le système doit être expert en auto-protection :** il doit être en mesure de détecter, identifier et se protéger des attaques de différents types pour maintenir la sécurité et l'intégrité de l'ensemble du système ;
- **le système connaît son environnement et agit en fonction :** les systèmes sont capables de négocier les uns avec les autres, pour notamment tirer parti de toutes les ressources disponibles ou sous utilisées ;
- **le système ne fonctionne pas en vase clos :** le système doit fonctionner dans un monde hétérogène en utilisant des standards ouverts pour s'intégrer le plus facilement possible dans son environnement ;
- **le système doit anticiper de manière transparente les situations :** il doit être capable de prévoir les besoins futurs et, dans la mesure du possible, de réserver des ressources pour la satisfaction de ceux-ci. Le système doit donc être capable d'anticipation.

3.1.2 Définitions de l'informatique autonomique

Deux ans après le discours et le manifeste de P. Horn, J. O. Kephart et D. Chess ont publié leur vision du chemin à suivre pour structurer l'informatique autonomique [KC03]. Pour les auteurs, l'essence de l'informatique autonomique est l'auto-gestion.

> "The essence of autonomic computing systems is self-management, the intent of which is to free system administrators from the details of system operation and maintenance and to provide users with a machine that runs at peak performance 24/7." [KC03]

Cette définition, largement citée, est tout à fait en ligne avec les huit caractéristiques de P. Horn décrites précédemment. Elle met en avant l'allègement des tâches pesant sur les épaules des administrateurs. Une seconde définition nous permet de repositionner le rôle de ceux-ci :

> "The Autonomic Computing Paradigm has been inspired by the human autonomic nervous system. Its overarching goal is to realize computer and software systems and applications that can manage themselves in accordance with high-level guidance from humans." [PH05]

D'après M. Parashar et S. Hariri, l'ambition de l'informatique autonomique n'est pas de remplacer les administrateurs, mais de leur donner un nouveau rôle, moins consommateur en temps et de plus haut niveau. La nouvelle tâche qui leur est dévolue, est donc de donner au système des objectifs de haut niveau, tout en laissant à ce dernier le soin de tenir ces objectifs dans un processus d'optimisation continu.

De plus, l'ambition de l'informatique autonomique est de proposer une approche globale afin de gérer les systèmes dans leur ensemble [Hor01, KC03]. En cela, cette approche cherche à aller au-delà de solutions existantes spécialisées dans la gestion d'aspects particuliers des systèmes (comme, par exemple, l'optimisation de la taille d'un *pool* de *threads*). L'objectif de l'informatique autonomique est donc de rendre l'ensemble du système autogéré.

3.1.3 Niveaux de maturité d'un système autonomique

A travers les définitions du paragraphe précédent, nous pouvons constater que l'ambition de l'informatique autonomique est extrêmement importante. IBM était pleinement conscient de cela et a donc proposé une approche incrémentale afin de permettre une transition plus aisée vers des systèmes autonomiques. Cette démarche est synthétisée dans un modèle, nommé *autonomic computing adoption model* [IBM06], représenté par la figure 3.1.

Ce modèle présente le niveau de maturité d'un système cherchant à tendre vers l'autonomie, en représentant celui-ci sur trois dimensions orthogonales que nous allons maintenant présenter.

La dimension fonctionnelle, sur l'axe horizontal caractérise le niveau d'automatisation sur une échelle présentant cinq niveaux de complexité croissante [GC03] :

- **le niveau basique** représente le point de départ des systèmes actuels. Chaque élément est géré de manière indépendante et manuellement par des administrateurs qui les installent, les configurent et, éventuellement, les remplacent ;
- **le niveau géré**, ajoute des outils de supervision, afin d'agréger les mesures des différents systèmes sur un ensemble de tableaux de bord, permettant ainsi de réduire le temps de collecte et de synthèse des informations ;

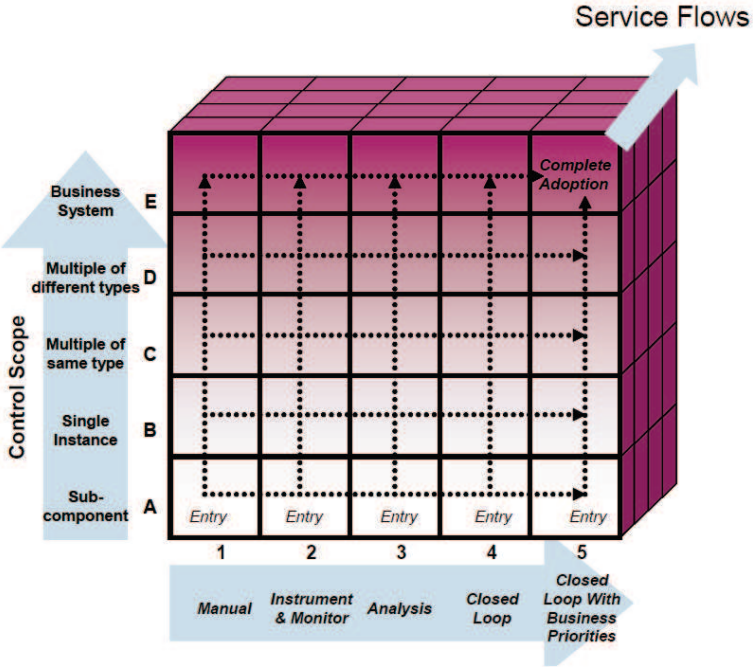

FIGURE 3.1 – Autonomic computing adoption model [IBM06].

- **le niveau prédictif** cherche à analyser les données et à les corréler selon des patrons afin de déterminer la configuration optimale et proposer des actions aux administrateurs qui restent seuls juges de leur mise en place ;
- **le niveau adaptatif** permet aux systèmes de décider eux-mêmes des actions à appliquer, sur la base d'informations sur leur état et leur contexte d'exécution ;
- **le niveau autonome** offre la possibilité aux administrateurs de guider le processus d'adaptation avec des politiques et des objectifs métiers qui peuvent être mis à jour dynamiquement.

L'axe vertical permet de prendre en compte la granularité des éléments gérés. Cet axe comprend lui aussi cinq niveaux de granularité croissante :

- **au niveau sous-composant** : seules certaines portions de ressources sont gérées, comme certaines applications hébergées sur un serveur ;
- **au niveau instance simple** : une ressource est gérée dans son intégralité comme, par exemple, un serveur complet ;
- **au niveau instances multiples** : un ensemble de ressources homogènes est géré comme, par exemple, une grappe de serveurs ;
- **au niveau instances multiples de différents types** : des ressources hétérogènes sont gérées en un tout comme, par exemple, un ensemble de serveurs, routeurs, bases de données ;

- **au niveau métier** : un ensemble complet de ressources logicielles et matérielles est géré selon une perspective de haut niveau, centrée sur le besoin métier traité par ces ressources.

La dernière dimension prend en compte l'automatisation des activités de gestion du système d'information comme la gestion des changements, des incidents... Les différentes activités peuvent présenter différents niveaux de maturité dans les dimensions présentées précédemment.

Selon [IBM06], la maturité d'un système autonomique peut donc évoluer selon trois axes :

- le niveau d'automatisation ;
- la granularité des éléments gérés ;
- l'automatisation des activités de gestion du système d'information.

3.1.4 Synthèse

Dans un contexte où la complexité des systèmes ne cesse de croître, les administrateurs ont de plus en plus de difficultés à appréhender les systèmes dans leur ensemble et à évaluer les conséquences d'une modification avant d'appliquer celle-ci. De plus, les adaptations doivent parfois être extrêmement rapides. P. Horn a donc proposé comme réponse l'informatique autonomique, dans laquelle le système est autogéré, surveillé et adapté en permanence afin de garantir un fonctionnement optimal, masquant ainsi sa complexité aux administrateurs tout en les libérant des opérations répétitives de maintenance.

Les administrateurs disposent ainsi d'un nouveau rôle, moins consommateur en temps et de plus haut niveau. La nouvelle tâche qui leur est dévolue, est de donner au système des objectifs de haut niveau, tout en laissant à ce dernier le soin de tenir ceux-ci dans un processus d'optimisation continu.

L'ambition de l'informatique autonomique étant de rendre le système autogéré dans son ensemble, IBM a proposé un modèle qui définit la maturité d'un système autonomique sur trois axes orthogonaux. L'idée sous-jacente est d'utiliser celui-ci afin de passer progressivement d'un système non autonomique à un système pleinement autonomique en améliorant pas à pas l'autonomie du système selon chacune de ces dimensions.

Pour cela, l'informatique autonomique s'est inspirée et appuyée de très nombreuses disciplines que nous allons présenter dans les sections suivantes.

3.2 Sources d'inspiration

Une originalité forte de l'informatique autonomique est de se présenter comme la confluence et le prolongement d'un ensemble de travaux préexistants, comme le montre cette définition de H. A. Müller :

> "Autonomic computing is not a new field but rather an amalgamation of selected theories and practices from several existing areas including control theory, adaptive algorithms, software agents, robotics, fault-tolerant computing, distributed and real-time systems, machine learning, human-computer interaction (HCI), artificial intelligence, and many more." [MOKW06]

Nous allons présenter ces différentes influences, ce qui permettra de mieux comprendre l'approche spécifique de l'informatique autonomique.

3.2.1 La biologie

Le terme d'informatique autonomique provient de cette première source d'influence qu'est la biologie. En anglais, le terme *autonomic* qualifie le système nerveux autonome, responsable de la régulation des fonctions vitales dans le corps humain. Ce système, non soumis au contrôle de la volonté a pour fonction de maintenir les paramètres vitaux de l'organisme humain (pression artérielle, digestion, glycémie, température...) à des valeurs permettant au corps humain un fonctionnement optimal. Pour cela, les changements à l'intérieur et à l'extérieur du corps humain sont surveillés et le système nerveux autonome réagit à chaque perturbation afin de maintenir le corps à un équilibre interne [PH05]. IBM note tout de même une différence importante entre le système nerveux autonome et les systèmes autonomiques [IBM06] : la plupart des décisions prises par le système nerveux sont involontaires. Au contraire, les systèmes autonomiques assurent des tâches que les administrateurs ont choisi de leur déléguer.

Au-delà de cette première influence de la biologie, les chercheurs se sont inspiré des mécanismes présents dans la nature [HS12, GSRU07] pour développer des modèles informatiques. Il s'agit des algorithmes biomimétiques. Par exemple, des algorithmes se sont inspirés du brassage génétique de la reproduction sexuée afin d'optimiser les multiples paramètres de systèmes complexes.

Un autre sous domaine très étudié des algorithmes biomimétiques concerne les algorithmes socio-mimétiques. Ceux-ci mettent en œuvre la collaboration d'un nombre important d'acteurs très spécialisés et au comportement très simple, ayant une connaissance limitée de leur environnement. De la somme de ces interactions simples à l'échelle microscopique résulte un comportement complexe à l'échelle macroscopique. On parle alors d'intelligence collective. L'exemple le plus fréquemment cité est celui des colonies de fourmis déposant des phéromones pour calculer les plus courts chemins [CDM91]. D'autres travaux s'inspirent, par exemple, des termites [Gra59] ou des abeilles [SS05].

Ces différents algorithmes peuvent ensuite être implantés dans le cadre de l'informatique multi-agents, dans laquelle un agent possède les quatre caractéristiques suivantes [WJ95] :

- **l'autonomie :** les agents évoluent de manière indépendante et sans contrôle extérieur sur leur comportement ou leur état ;
- **la sociabilité :** les agents interagissent les uns avec les autres (et, potentiellement, aussi avec des humains) à l'aide d'un langage de communication ;
- **la réactivité :** les agents perçoivent leur environnement et réagissent à son évolution ;
- **la pro-activité :** les agents ne font pas que répondre aux modifications de l'environnement, ils peuvent aussi prendre des initiatives pour atteindre un but donné.

L'approche multi-agents est utilisée dans de nombreux projets en informatique autonomique comme, par exemple, [TCW+04] et [DWH03].

3.2.2 La théorie du contrôle

Une seconde source d'influence est la théorie du contrôle qui a contribué à l'informatique autonomique avec un apport capital : la boucle de rétroaction [SEMD10]. L'objectif de celle-ci est de réguler la valeur d'une grandeur physique (telle que la vitesse ou la température) pour la maintenir dans un intervalle de valeurs donné.

Une boucle de rétroaction est composée de trois éléments [DFT92] (figure 3.2) : le système que l'on désire réguler, un ensemble de sondes pour connaître l'état de ce système et un contrôleur qui calcule une consigne à appliquer en fonction de la différence entre l'état souhaité et l'état actuel du système.

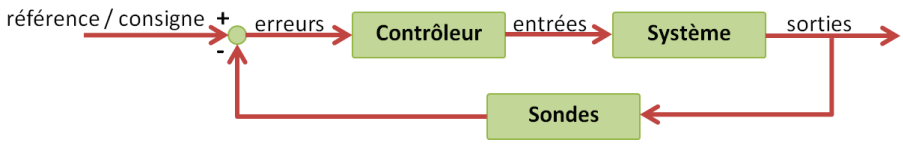

FIGURE 3.2 – Boucle de rétroaction.

Ce concept de boucle a donc été largement repris par l'informatique autonomique [MH04]. Sur la figure 3.2, nous pouvons voir un système pris en charge par un gestionnaire autonomique. Ce gestionnaire perçoit l'état du système géré par l'intermédiaire de capteurs, qui correspondent aux sondes de la boucle de rétroaction. Il peut ensuite raisonner comme le fait un contrôleur pour bâtir une stratégie et appliquer celle-ci au système à l'aide d'effecteurs.

3.2.3 Autres sources d'inspiration

L'informatique autonomique s'est aussi inspirée de nombreuses autres disciplines. C'est le cas notamment de l'intelligence artificielle, dont le but est de construire des machines reproduisant le niveau d'intelligence de l'homme [Bro91, RNC+95]. Logiquement, cette discipline a donc aussi été une source importante d'inspiration pour l'informatique autonomique [AEW03], même si son ambition dépasse de loin les objectifs de cette dernière. En effet, l'objectif de l'informatique autonomique est beaucoup plus modeste ; car il vise « simplement » à ajouter des capacités d'auto-adaptation à des systèmes informatiques. De plus, l'intelligence n'est pas ici un but en soi, comme dans le cadre de l'intelligence artificielle, mais un moyen pour parvenir à l'administration de systèmes complexes. Enfin, les systèmes autonomiques restent subordonnés à un administrateur humain qui définit des buts, ce qui n'est pas nécessairement le cas d'un système intelligent.

La théorie des fonctions d'utilité est aussi utilisée très largement dans les systèmes autonomiques [KD07]. Elle propose d'optimiser les paramètres d'un système en utilisant une fonction d'utilité. Celle-ci attribue une valeur d'utilité (valeur scalaire) qui représente la pertinence des paramètres fournis pour prendre en charge le contexte d'exécution courant. Il est ainsi possible de sélectionner le plan d'action le plus adéquat [RK76].

Dans un environnement d'exécution incertain, le domaine des probabilités apporte des outils permettant de bâtir des gestionnaires autonomiques probabilistes. A titre d'exemple, H. Guo [Guo03] utilise des réseaux bayésiens pour sélectionner automatiquement le meilleur algorithme sur la base des caractéristiques du problème à résoudre.

La robotique a aussi contribué de manière significative. En effet, les robots doivent souvent évoluer dans des environnements ouverts et très dynamiques, dans lesquels le besoin d'adaptation au contexte est important. L'informatique autonomique a donc puisé dans les travaux de cette discipline, notamment concernant la planification en milieu incertain.

L'économie [KC03] est elle aussi une source d'inspiration pour ses processus internes de régulation, même si depuis la crise de 2008, nous pouvons douter de la complète rationalité des systèmes économiques à se maintenir à un point optimal. Des travaux se sont notamment intéressés à la négociation de ressources sur la base de leurs coûts d'utilisation [BAGS02].

Enfin, le génie logiciel a apporté des bonnes pratiques afin d'implanter des systèmes autonomiques en proposant des architectures qui permettent notamment de séparer correctement les différentes préoccupations afin de mieux les traiter. A ce sujet, nous présenterons l'architecture proposée par IBM dans la suite de cette thèse (section 3.4 page 73).

3.2.4 Synthèse

Compte tenu de l'ambition importante de l'informatique autonomique, ce domaine s'est développé et structuré en s'appuyant sur les résultats obtenus dans de nombreuses disciplines. C'est qui explique la remarque de H. A. Müller, qui considère l'informatique autonomique comme la confluence et le prolongement de travaux préexistants plutôt que comme un nouveau champ d'étude [MOKW06].

En plus de donner son nom à l'informatique autonomique par une analogie avec le corps humain, la biologie a contribué en apportant des mécanismes issus de la nature, implantés dans le cadre des algorithmes biomimétiques qui font massivement recours à l'informatique multi-agent.

La théorie du contrôle a aussi contribué de manière capitale avec la boucle de rétroaction, qui a fait l'objet de modélisation mathématique de longue date. Les systèmes autonomiques se sont largement inspirés de ces résultats, comme dans l'architecture proposée par IBM.

L'intelligence artificielle a apporté aux systèmes autonomiques des capacités de raisonnement, même si l'ambition de l'informatique autonomique est moindre dans la mesure où l'intelligence artificielle est dans ce cadre un outil, et non un but en soi.

D'autres travaux issus de la robotique, comme la planification ou les fonctions d'utilité, ont aussi inspiré l'informatique autonomique. C'est aussi le cas de l'économie et du génie logiciel indispensables pour réaliser de manière rigoureuse toute implantation de systèmes autonomiques.

Compte tenu des ambitions fortes de l'informatique autonomique et des multiples domaines qui l'ont influencée, l'autogestion des systèmes peut être considérée sous un nombre très important de facettes. Celles-ci ont fait l'objet de nombreuses études et ont été classées sous la forme de propriétés. Nous allons détailler à présent les plus connues et étudiées d'entre elles.

3.3 Les propriétés d'un système autonomique

3.3.1 Propriétés de base

L'autogestion des systèmes proposée par l'informatique autonomique peut être perçue à travers de très nombreuses propriétés. Cependant, il en existe quatre principales, proposées initialement par IBM et détaillées par la suite dans [KC03] et [BBC+03] :

- **l'auto-configuration** (*self-configuring*) prend en charge la configuration du système, sur la base de buts de haut niveau représentant des objectifs métiers. De plus, quand un composant est ajouté, il est configuré de façon transparente et le reste du système s'adapte à sa présence.
- **l'auto-optimisation** (*self-optimizing*) prend en charge l'ajustement permanent des paramètres de configuration pour optimiser les performances des systèmes dans un environnement en constante évolution. Pour cela, des algorithmes d'apprentissage permettent d'améliorer les réglages au fur et à mesure du temps et des essais successifs.
- **l'auto-réparation** (*self-healing*) permet de détecter, diagnostiquer et réparer les problèmes causés par un dysfonctionnement ou une défaillance matérielle. En effet, puisqu'il n'est pas possible d'anticiper toutes les erreurs, cette propriété vise à leur correction. Notons que la tolérance aux fautes est un aspect de l'auto-réparation [SS98].
- **l'auto-protection** (*self-protecting*) vise à défendre le système contre des attaques extérieures ou des défaillances en cascade qui n'ont pu être corrigées par des mesures d'auto-réparation. L'auto-protection vise aussi l'anticipation des dysfonctionnements en engageant au plus tôt des procédures pour les éviter ou en diminuer la portée.

Ces quatre propriétés de base des systèmes autonomiques sont parfois désignées sous l'acronyme CHOP (*Configuring, Healing, Optimizing, Protecting*) [Mil05].

3.3.2 Arbre de R. Sterritt et D. Bustard

Pour situer les quatre propriétés de base définies par IBM dans le large contexte de l'informatique autonomique, R. Sterrit et D. Bustard ont proposé un arbre de l'informatique autonomique [SB03a] (figure 3.3).

Cet arbre est composé de quatre ramifications. La première permet d'exprimer le but de l'informatique autonomique : l'auto-gestion des systèmes. La seconde indique quatre objectifs, qui correspondent aux quatre propriétés définies par IBM (voir paragraphe précédent).

Une troisième ramification [SB03b] fournit quatre attributs :

- **l'auto-connaissance** (*self-aware*) : le système doit posséder des capacités d'introspection qui lui permettent d'avoir conscience de ses composants et de son état interne ;
- **la connaissance de l'environnement** (*environment aware*) : le système doit disposer d'une connaissance de son environnement d'exécution ;
- **l'auto-surveillance** (*self-monitoring*) : le système doit détecter aussi vite que possible les évolutions du contexte ou de son état interne qui doivent entraîner une adaptation ;
- **l'auto-réglage** (*self-adjusting*) : c'est la capacité qu'a le système de se modifier lui-même.

Enfin, une quatrième et dernière ramification trace la voie pour faire de l'informatique autonomique une réalité [Ste02] :

- **la première étape est de commencer à construire des systèmes autonomiques**, en recourant du génie logiciel. C'est en faisant que l'on découvrira les bons patrons ;
- **la seconde étape, à plus long terme, est de parvenir à créer des systèmes pleinement autonomiques**. Cela nécessitera le recours à un ensemble de disciplines que nous avons décrit précédemment et, notamment, l'intelligence artificielle qui apportera les algorithmes d'auto-apprentissage.

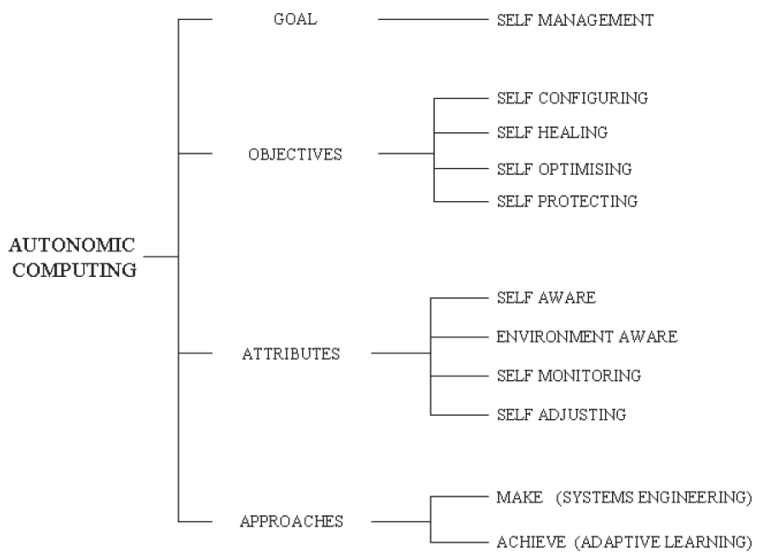

FIGURE 3.3 – Arbre de l'informatique autonomique [SB03a].

3.3.3 Synthèse

L'autogestion des systèmes proposée par l'informatique autonomique peut être perçue à travers de très nombreuses propriétés. Cependant, il en existe quatre principales, proposées initialement par IBM : l'auto-configuration, l'auto-optimisation, l'auto-réparation et l'auto-protection. Celles-ci ont été largement étudiées et discutées, notamment par R. Sterrit et D. Bustard qui les ont positionnées par rapport à l'état de l'art des travaux de l'informatique autonomique [SB03a].

De plus, ces quatre propriétés proposées par IBM ont servi de base à des listes de taille plus conséquentes qui se sont développées au fur et à mesure de l'apparition de nouveaux besoins [Ste05, Tia03, DWH06].

A présent que nous avons défini l'informatique autonomique et que nous avons décrit ses propriétés de base, nous allons nous intéresser aux architectures logicielles pour bâtir ces systèmes. En effet, compte tenu de la complexité de ceux-ci, le recours à des pratiques rigoureuses de génie logiciel est une nécessité. En particulier, il est capital de disposer d'une approche architecturale qui sépare de manière stricte les différentes préoccupations afin de les traiter de manière indépendante et de simplifier ainsi la compréhension et l'évolution du système.

C'est pour répondre à cette préoccupation de génie logiciel que IBM a adossé à sa vision théorique une proposition d'architecture logicielle afin d'implanter les systèmes autonomiques, que nous présentons dans la partie suivante.

3.4 Architecture des systèmes autonomiques selon IBM

3.4.1 Architecture des systèmes autonomiques

Les systèmes autonomiques étant des systèmes complexes, le besoin de disposer de guides pour les bâtir est apparu très rapidement. C'est pour répondre à ce besoin que IBM a proposé une architecture de référence [IBM06]. Dans celle-ci, un système autonomique est composé à partir d'un ou plusieurs éléments autonomiques en collaboration. Ces différents sous-systèmes peuvent être agencés selon plusieurs topologies. Les trois principales sont les suivantes :

- **les architectures centralisées** utilisent un seul élément autonomique pour gérer l'ensemble du système. Cette approche permet une plus grande cohérence dans les décisions prises pour administrer le système, mais se heurte en retour à une plus grande complexité de réalisation et de modification, si le système est de grande traille.
- **les architectures distribuées [WSG+13]** utilisent un ensemble d'éléments autonomiques qui interagissent et se coordonnent selon l'approche des systèmes multi-agents [KC03, TCW+04]. L'idée centrale est de gérer un système complexe grâce à des agents au comportement unitairement simple, donc plus facile à implanter. De plus, le passage à l'échelle se trouve facilité par le découpage du système mis en œuvre. Cette approche, séduisante en théorie, se heurte en pratique à la difficulté de faire émerger au niveau du système le comportement souhaité, qui peut être très imprévisible et très instable. Cela est souvent causé par des objectifs contradictoires qui peuvent être poursuivis par des agents.
- **les architectures hiérarchiques** sont une variante des architectures distribuées. Celles-ci adoptent une organisation pyramidale, qui permet de garantir une cohérence d'action entre les différents agents qui reçoivent les consignes de leur supérieur hiérarchique. Le principal inconvénient de cette approche est le nombre de communications qui augmente nécessairement afin de transmettre les ordres et de contrôler les résultats obtenus.

Nous allons à présent détailler l'architecture des éléments autonomiques. Ensuite, nous détaillerons l'architecture proposée par IBM pour implanter la partie qui gère le processus d'adaptation : le gestionnaire autonomique.

3.4.2 Architecture des éléments autonomiques

Présentation

Un élément autonomique [KC03] est constitué par deux parties : un système géré et un gestionnaire autonomique, le second ayant la charge d'administrer le premier. Pour cela, l'administrateur donne des politiques au gestionnaire autonomique qui interagit avec le système géré par l'intermédiaire de points de contrôle (capteurs et actionneurs). De plus, on qualifie de boucle autonomique l'ensemble constitué par le gestionnaire autonomique et

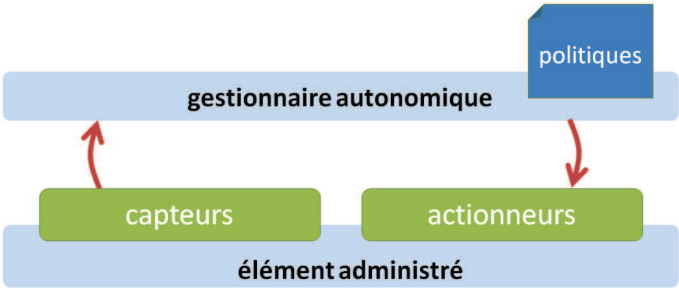

FIGURE 3.4 – Elément autonomique.

les points de contrôle du système administré. Nous allons à présent détailler les différentes parties d'un élément autonomique, représenté sur la figure 3.4.

Elément administré

Un élément administré, aussi appelé ressource supervisée ou système géré, est la ressource qui est prise en charge par le gestionnaire autonomique. Il s'agit d'un composant logiciel (d'un simple élément à une application de grande taille) ou matériel (disque dur, interface réseau...). Celui-ci doit être configuré manuellement lorsqu'il n'est pas pris en charge par un gestionnaire autonomique.

Points de contrôle

Les points de contrôle sont des interfaces de l'élément administré qui lui permettent d'être accessible par « le reste du monde ». Pour J. O. Kephart [Kep05], c'est même la perspective d'administration d'un système par un gestionnaire autonomique qui va indiquer avec quels capteurs et effecteurs le système doit être conçu. Ces points de contrôle sont de deux types : capteurs et actionneurs.

- **Les capteurs (ou sondes)** collectent des informations sur l'élément administré. Ces informations peuvent être de haut niveau (comme la supervision d'un ensemble de serveurs [RBC05]) ou au contraire de granularité très fine (comme le taux de remplissage d'un *buffer* [Mor13]).
- **Les actionneurs (ou effecteurs)** permettent d'agir sur le système à différents niveaux d'abstraction. A un haut niveau, il peut, par exemple, s'agir de la modification du nombre de serveurs utilisés pour un traitement et, à bas niveau, de la mise à jour de paramètres de configuration [SSB05].

Gestionnaire autonomique

Une des spécificités de l'approche proposée par IBM est de séparer strictement le gestionnaire autonomique de l'élément administré. Cela permet de dissocier clairement le code applicatif (l'élément administré) du code d'adaptation (gestionnaire autonomique). De cette manière, une séparation stricte des préoccupations est effectuée, ce qui augmente la flexibilité, la maintenabilité et la réutilisabilité des différents éléments [GSC09]. De plus, cela permet d'utiliser des technologies différentes pour chaque élément. Cette séparation est donc

aujourd'hui reconnue comme une bonne pratique de génie logiciel pour l'implantation des systèmes autonomiques.

Le gestionnaire autonomique a donc la charge de surveiller un sous-système via les capteurs, d'analyser son état pour le corriger si besoin à l'aide d'une série d'actions transmises par les actionneurs. Cette tâche nécessite des algorithmes d'analyse et de décision qui peuvent être de taille extrêmement variable. Dans les cas simples, un simple bloc de code monolithique peut suffire. Dans des cas plus complexes, une architecture de grande taille est nécessaire. Pour celles-ci, IBM a proposé l'architecture MAPE-K détaillée dans la partie suivante.

Politiques

Les politiques sont fournies par l'administrateur (ou par un autre gestionnaire autonomique). Elles guident le gestionnaire autonomique dans le choix des actions à entreprendre pour réagir aux évolutions de son environnement et aux modifications de l'état de la ressource gérée. Elles expriment ce que l'on souhaite obtenir (le quoi) plutôt que la manière de l'obtenir (le comment). Elles doivent pouvoir être définies clairement et sans erreur, sans quoi les conséquences lors de l'exécution du système pourraient être catastrophiques. Pour cette raison, les systèmes autonomiques peuvent se protéger eux-mêmes des politiques incohérentes, dangereuses ou impossibles à réaliser.

Ces politiques doivent donc être de haut niveau. Leur expression, pour les rendre intelligibles par une machine est un sujet de recherche crucial dans le cadre de l'informatique autonomique. Dans ce cadre, trois approches sont aujourd'hui largement utilisées [KW04] :

- **les politiques d'action** indiquent l'action qui doit être entreprise quand un système atteint un état prédéterminé. Ces politiques sont souvent exprimées à l'aide de règles ECA (Evénement, Condition, Action). Chaque règle est conçue comme un tuple, où l'événement déclenche la vérification d'une condition. Si celle-ci est satisfaite, l'action correspondante est entreprise.
- **les politiques de but** : au lieu de spécifier exactement les actions à entreprendre dans chaque situation, les politiques de but expriment le ou les états souhaités pour le système ou un critère de validité de l'état global du système. C'est ensuite au gestionnaire autonomique de déterminer la manière d'atteindre un état valide. Les politiques de but sont bien adaptées quand le système dispose d'un très grand nombre d'états possibles. Elles peuvent souvent être exprimées à un haut niveau d'abstraction, ce qui simplifie leur définition et leur évolution.
- **les fonctions d'utilité** [WTKD04] permettent d'évaluer un ensemble d'états en leur attribuant une note. De cette manière, l'état le mieux évalué peut être choisi et un processus d'adaptation peut être mis en œuvre pour l'atteindre. Les fonctions d'utilité permettent ainsi de sélectionner l'état le plus approprié, là où les politiques de buts ne cherchent qu'à atteindre un état « acceptable ». Cependant, les fonctions d'utilité demandent une évaluation de chaque état possible pour le système, ce qui peut ne pas être possible dans la pratique.

Quel que soit le type de politique employée, l'intégration de celles-ci peut se faire de deux manières différentes. La première est d'implanter celles-ci directement dans le code. La seconde est de les séparer du code en les exprimant avec un DSL. La seconde solution, plus flexible, est un peu plus complexe à mettre en œuvre.

3.4.3 Architecture des gestionnaires autonomiques

Pour guider l'implantation des gestionnaires autonomiques, IBM a proposé une architecture détaillée dans un livre blanc [IBM06]. Elle est composée de quatre fonctions qui partagent une base de connaissances (figure 3.5) :

- **la fonction de supervision** collecte, agrège et filtre les données en provenance de l'élément administré via ses capteurs. La transmission des données à la fonction d'analyse peut être effectuée périodiquement ou quand la fonction de surveillance le juge utile. Dans ce second cas, une analyse plus poussée sur les données doit donc être effectuée.
- **la fonction d'analyse** modélise la situation pour détecter la présence de problèmes. Pour cela, elle tient compte de l'état courant, mais aussi des états antérieurs. Notons que cette fonction peut avoir un rôle prédictif et donc pronostiquer un dysfonctionnement futur, notamment en se basant sur des tendances observées sur une fenêtre temporelle.
- **la fonction de planification** utilise la liste de problèmes transmise par la fonction d'analyse pour définir un plan d'actions. Celui-ci a pour but de modifier l'état de l'élément administré pour le corriger ou lui permettre d'affronter plus efficacement une situation future.
- **la fonction d'exécution** exécute un plan d'actions défini par la fonction de planification. Pour cela, elle utilise les actionneurs de l'élément administré.
- **la base de connaissances** est partagée par les quatre fonctions ci-dessus. Elle peut être mise à jour en permanence par n'importe laquelle de ces quatre fonctions, à des fins de stockage et d'historisation. De plus, elle permet de stocker les politiques et les objectifs de haut niveau.

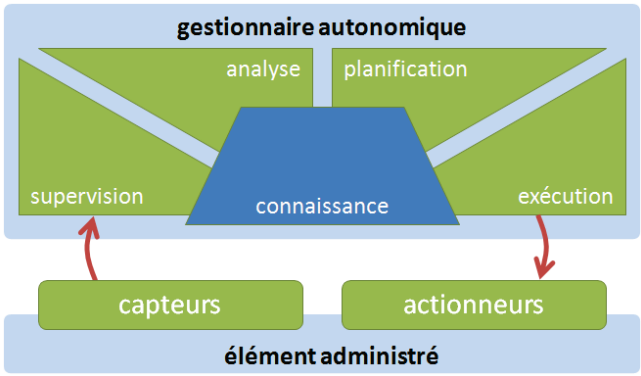

FIGURE 3.5 – Boucle de contrôle MAPE-K.

Cette architecture est connue sous le nom de MAPE-K (*Monitor, Analize, Plan, Execute, Knowledge*). Outre une séparation des préoccupations pour l'implantation du gestionnaire autonomique, cette architecture modulaire offre une perspective de réutilisabilité des éléments qui la composent. De plus, cela permet d'utiliser des outils ou des technologies différentes pour chaque fonction, en faisant collaborer celles-ci par l'intermédiaire d'interfaces standardisées. Un autre avantage de cette décomposition est la possibilité de distribuer les fonctions sur des plates-formes différentes. Dans ce cas, il faut veiller à ce que les délais induits par les transmissions ne nuisent pas à la réactivité globale du système.

Notons que nous avons présenté ici un gestionnaire autonomique séparé de l'élément administré. Ce n'est pas forcément le cas, mais cette dissociation entre le système fonctionnel et la logique d'adaptation offre de nombreux avantages [GSC09, Bou08, Mau10]. Tout d'abord, l'adaptation dans une entité séparée est plus facilement modifiable et extensible. De plus, séparer la logique d'adaptation permet d'appliquer la boucle MAPE-K à des systèmes préexistants sans les modifier et sans accéder au code source, pour peu que ceux-ci disposent des capteurs et d'actionneurs nécessaires. Enfin, cette séparation quand elle est mise en œuvre avec des points de configuration, peut permettre la réutilisation du gestionnaire autonomique entre plusieurs systèmes.

Il est important de garder à l'esprit que l'architecture MAPE-K propose un cadre général de décomposition pour la construction d'applications autonomiques. Elle n'apporte pas de solution technique d'implantation [LDM13]. De plus, cette proposition peut être adaptée selon les systèmes. Par exemple, il peut être parfois judicieux de fusionner les fonctions de supervision et d'analyse pour certains gestionnaires autonomiques de petite taille.

3.4.4 Synthèse

Pour simplifier la conception des systèmes autonomiques, IBM a proposé une architecture de référence [IBM06]. Dans celle-ci, un système autonomique est composé à partir d'un unique élément autonomique (on parle alors d'architecture centralisée) ou de plusieurs en collaboration, formant ainsi une architecture distribuée, qui peut éventuellement être hiérarchique.

Chaque élément autonomique est constitué de deux parties : le gestionnaire autonomique et l'élément administré. Cela permet de dissocier clairement le code applicatif (l'élément administré) du code d'adaptation (gestionnaire autonomique). Le gestionnaire autonomique a la charge de surveiller un système via les capteurs, d'analyser son état pour le corriger si besoin à l'aide d'une série d'actions transmises à l'élément administré par les actionneurs. Pour guider son action, le gestionnaire autonomique se base sur un ensemble de politiques fournies par un administrateur.

En raison de la complexité d'implantation des gestionnaires autonomiques, IBM a proposé une architecture appelée MAPE-K qui guide la conception des gestionnaires autonomiques en séparant différentes préoccupations. Cette architecture MAPE-K est composée de quatre fonctions (supervision, analyse, planification, exécution) qui partagent une base de connaissances.

L'un des points clé de l'architecture MAPE-K est cette connaissance. Grâce à celle-ci, le système peut être analysé plus finement en donnant, par exemple, davantage de sens aux informations remontées par la fonction de supervision. De plus, la connaissance peut aussi permettre à la fonction de planification de s'appuyer sur les résultats de précédentes adaptations.

Parmi l'ensemble des utilisations possibles de cette connaissance, l'une d'entre elles nous apparaît particulièrement intéressante. Il s'agit de la modélisation de l'architecture exécutée à partir des données remontées au gestionnaire autonomique. Cela permet de disposer des avantages liés aux architectures et, notamment, un point de vue de haut niveau sur l'ensemble du système. Il n'est donc pas surprenant que de nombreuses approches se soient appuyées sur les architectures afin de représenter la connaissance liée à l'exécution.

Nous allons donc à présent nous intéresser à ces architectures logicielles et à leur participation à la base de connaissances dans le cadre de l'architecture MAPE-K d'IBM.

3.5 Architectures pour représenter la connaissance

3.5.1 Introduction et critères de caractérisation

Nous avons vu précédemment que la question de la gestion de la connaissance est centrale dans l'informatique autonomique. En fait, cette préoccupation est même cruciale dans le domaine plus large des systèmes auto-adaptables, dans lequel l'informatique autonomique n'est qu'une approche parmi d'autres, même si elle occupe aujourd'hui une place prépondérante.

Dans le cadre des systèmes auto-adaptables, l'intérêt d'utiliser un modèle architectural qui représente le système administré lors de son exécution a été mis en lumière extrêmement tôt [OMT98, OGT+99]. En effet, un modèle architectural qui évolue simultanément au système administré pour en refléter l'état offre de multiples avantages [GCH+04, GSC09]. Tout d'abord, en tant que modèle abstrait, il fournit une perspective globale sur le système et permet de mettre en avant les propriétés et comportements importants du système. De plus, il permet de rendre explicites les contraintes d'intégrité d'un système ; cela peut permettre d'établir le périmètre des modifications autorisées et de s'assurer de la validité d'un changement. Enfin, un modèle peut être associé avec des contraintes architecturales qui, par exemple, interdisent des cycles dans l'architecture.

Nous allons à présent nous pencher sur un ensemble de travaux, qui utilisent des architectures dans le cadre du processus d'adaptation. Chaque approche sera évaluée selon les critères suivants :

- **expression de contraintes de conception :** est-il possible de définir des contraintes architecturales comme, par exemple, des invariants qui doivent être respectées à tout moment ? Comment ?
- **réification de l'architecture de conception :** quelle partie du système s'occupe de fournir un modèle de l'architecture exécutée ? Comment est exprimé celui-ci ?
- **vérification de la conformité de l'architecture :** comment est validée l'architecture, au regard des contraintes de conception exprimées ?
- **mécanisme de reconfiguration :** comment et par qui l'architecture exécutée est-elle mise à jour ?
- **plate-forme d'exécution :** quelle est la plate-forme qui porte le système administré ?
- **spécificité de l'approche :** quels sont les points d'attention particuliers qui ont été traités ?

Nos critères couvrant jusqu'à l'exécution des systèmes, nous ne traiterons pas des ADL (*Architecture Description Language*) [MT00, BCDW04], qui servent à décrire les architectures, et non à les exécuter, même si certaines approches que nous allons détailler utilisent certains d'entre eux.

Pour commencer l'évaluation des différentes approches, nous allons tout d'abord étudier le *framework* Rainbow, conçu par D. Garlan et S.-W. Cheng à partir du début des années

2000. En effet, celui-ci fait figure de pionnier ; nous lui accorderons donc une attention particulière. Les travaux suivants seront traités de manière moins développée.

3.5.2 Le *framework* Rainbow

3.5.2.1 Présentation

Rainbow [Che08] est un *framework* qui permet d'ajouter des capacités d'auto-adaptation à des systèmes existants. Pour cela, il offre un cadre pour superviser le système, le réifier dans un modèle architectural, détecter des améliorations, choisir un plan d'action et effectuer ces modifications. Le *framework* Rainbow fournit une infrastructure réutilisable avec des points de configuration clairement définis afin de l'adapter à des systèmes particuliers. De cette manière, le travail d'ingénierie se trouve allégé, structuré, et ainsi, simplifié. Les gains envisagés par ses auteurs sont les suivants [GSC09] :

- une réduction du coût des développements, en capitalisant sur une infrastructure réutilisable ;
- une simplification du travail des développeurs, qui n'ont plus qu'à se focaliser sur un ensemble restreint de petits incréments ;
- une plus grande clarté du code produit, qui sépare la logique d'adaptation dans des primitives écrites avec le langage Stich.

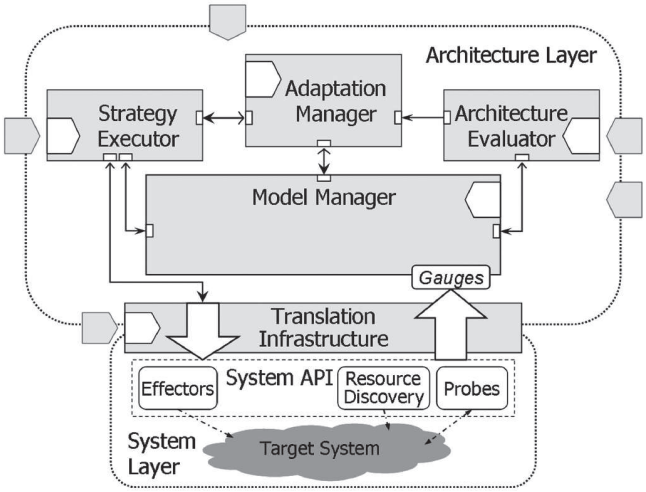

FIGURE 3.6 – Le *framework* Rainbow [GSC09].

Nous allons maintenant décrire la figure 3.6, qui présente en détails le *framework* Rainbow.

Pour interagir avec l'élément administré (*Target System*), Rainbow utilise des actionneurs nommés effecteurs (*Effectors*) et des capteurs nommés sondes (*Probes*). Ces points d'interaction étant de bas niveau, Rainbow propose l'utilisation d'une couche appelée *Translation Infrastructure*, qui utilise les travaux précédents de D. Garlan [GSC01] et S.-W. Cheng

[CHG+04]. Le premier rôle de cette couche est de faire remonter les informations en provenance du système administré. Pour cela, les mesures de bas niveau des sondes sont agrégées et corrélées par des jauges (*gauges*) afin de fournir des données de plus haut niveau. Symétriquement, la couche de traduction permet de piloter les modifications à appliquer au système à l'aide de primitives. Leur exécution peut nécessiter d'un simple appel de méthode à l'utilisation d'un sous-système de *workflow* complexe. Dans tous les cas, le système est modifié via les effecteurs.

Pour centraliser la connaissance du système administré et de son environnement, un gestionnaire de modèles (*Model Manager*) maintient à jour deux modèles, en s'appuyant sur les jauges présentées précédemment :

- le modèle architectural, qui représente l'élément administré ;
- le modèle de l'environnement, qui représente le contexte d'exécution, en y incluant la représentation des ressources nécessaires au bon fonctionnement du système.

Lors de chaque évolution des modèles, même minime, le vérificateur d'architecture (*Architecture Evaluator*) teste la conformité du modèle architectural par rapport à un ensemble de contraintes qui peuvent porter sur les deux modèles.

Si un problème est détecté, le gestionnaire d'adaptations (*Adaptation Manager*) va sélectionner une stratégie (définie à l'aide d'un DSL nommé Stitch [Che08]) pour corriger le système. Pour cela, le gestionnaire d'adaptations évalue les pré-conditions d'un ensemble de stratégies, pour ne considérer par la suite que celles qui sont pertinentes en fonction de l'état courant du modèle du système et du contexte. Les stratégies restantes sont ensuite évaluées à l'aide d'une fonction d'utilité. Celle qui obtient le meilleur score est alors sélectionnée.

L'orchestrateur d'exécution de stratégie (*Strategy Executor*) exécute la stratégie sélectionnée par le gestionnaire d'adaptations en lançant le script Stitch correspondant. Cela entraîne l'appel de primitives, qui pilotent les effecteurs du système au travers de l'infrastructure de traduction.

Les différents éléments que nous venons de décrire, correspondent aux quatre fonctions et à la base de connaissances de l'architecture MAPE-K. La connaissance est formalisée dans les modèles architecturaux, gérés par le gestionnaire de modèles. La fonction de supervision repose sur les sondes et les jauges. La fonction d'analyse est effectuée par l'évaluateur d'architecture, qui recherche les dysfonctionnements par l'analyse des modèles. La planification est prise en charge par le gestionnaire d'adaptation, qui décide de la stratégie à sélectionner. Enfin l'exécution est prise en charge par l'orchestrateur d'exécution de stratégie qui applique le plan d'action au système administré par l'intermédiaire des primitives.

A présent que nous disposons d'une vision globale du *framework* Rainbow, nous allons nous intéresser à une caractéristique clé de celui-ci : l'utilisation d'un style architectural.

3.5.2.2 Le style architectural

Il existe de nombreuses manières pour définir un modèle : réseaux de Pétri, équations différentielles... [SEMD10]. Dans l'approche proposée par Rainbow, un modèle architectural représente le système sous la forme d'un graphe de composants en interaction [CGS+02]. Les nœuds du graphe, appelés composants, représentent les éléments principaux du système. Les arcs, appelés connecteurs, représentent les interactions possibles entre composants. Cette description d'extrêmement haut niveau, à base de composants et de connecteurs permet de représenter une très vaste palette de systèmes. Cependant, par manque de détails, une telle abstraction est totalement inopérante : il est impossible de raisonner sur un système particulier, de vérifier sa cohérence et les modifications à lui apporter, et de faire le lien avec un système réel.

Pour cela, le *framework* Rainbow propose un système de configuration qui repose sur la notion de style architectural [CGS+02]. Celui-ci offre un jeu de règles de composition, de validation et de correction des éléments de l'architecture exécutée afin de fournir les connaissances nécessaires à l'administration d'un système particulier.

De manière générale, un style architectural dispose de quatre composantes principales [AAG93, AAG95] :

- **le vocabulaire** permet de nommer les différents types possibles d'éléments, que ce soit des types de composants (par exemple : base de données, client, serveur...), des types de connecteurs (SQL, HTTP, RCP...) ou des interfaces de l'un ou l'autre ;
- **les règles d'assemblage ou contraintes** définissent les configurations possibles à partir des composants et des connecteurs. Par exemple, il peut être imposé dans une architecture client/serveur que chaque client soit lié à un serveur ;
- **les propriétés** permettent de caractériser l'état d'un élément donné. Par exemple, un élément de type serveur pourra avoir une propriété exprimant son pourcentage de charge à chaque instant ;
- **les analyses** utilisent des éléments définis précédemment. Par exemple, des analyses de performance sur les systèmes client/serveur sont effectuées dans [SG98].

Dans le cadre de Rainbow, l'expression du style architectural met en jeu :

- **une grammaire**, qui définit les éléments gérés par le modèle, leurs liens, ainsi que la manière avec laquelle ils sont dépendants des valeurs fournies par les jauges ;
- **un ensemble de règles**, qui permettent de spécifier les limites de validité du système et/ou les états qui nécessitent une adaptation ;
- **un ensemble de stratégies et la fonction d'utilité**. De plus, il est nécessaire de définir les **variables** sur lesquelles la fonction d'utilité peut être appliquée, et le lien de ces variables avec le modèle. Il faut enfin prévoir des **attributs** qui permettent d'exprimer le rapport coût/bénéfice de chaque stratégie sur les variables qui servent à évaluer la fonction d'utilité.

En plus de ces quatre composantes que l'on retrouve fréquemment, D. Garlan a étendu la notion de style architectural avec une cinquième : les opérateurs.

- **les opérateurs** définissent les primitives qui peuvent être appliquées à un élément du système géré pour modifier sa configuration.

Pour illustrer le fonctionnement de Rainbow et sa configuration à l'aide d'un style architectural, nous allons détailler l'exemple de Znn.com proposé par les auteurs du framework.

3.5.2.3 L'exemple de Znn.com

L'exemple Znn.com est détaillé dans plusieurs publications [Che08, CGS09, CPGS09, GSC09]. Il s'agit d'un service inspiré du célèbre site d'information cnn.com. Il a pour but de fournir des contenus relatifs à l'actualité sur l'Internet. Le service Znn.com (figure 3.7) utilise un répartiteur de charge qui redirige les requêtes des clients (c0, c1, c2) sur une grappe de serveurs identiques (s0, s1, S2, s3), dont la taille peut être ajustée dynamiquement. Par mesure de simplicité, les requêtes des clients sont sans état. A chaque instant, il est possible de connaître la charge de chaque serveur, ainsi que la bande passante disponible sur chaque lien entre les serveurs et les clients.

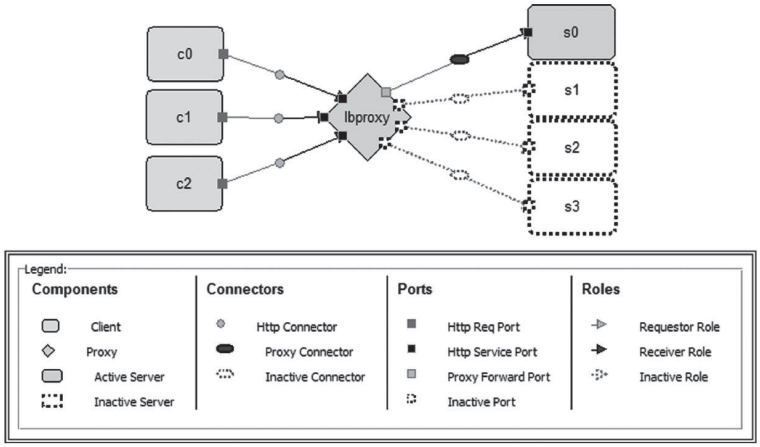

FIGURE 3.7 – Architecture de Znn.com [GSC09].

La contrainte appliquée au service Znn.com est qu'il soit en mesure de servir les clients dans (presque) n'importe quelles conditions de charge, tout en garantissant un temps de réponse raisonnable et en maintenant un nombre maximum de serveurs dans la grappe. Ponctuellement, en raison de sujet d'actualités populaires, le service Znn.com peut ne pas être en mesure d'offrir un temps de réponse satisfaisant à tous ses clients, même avec la taille maximale de la grappe de serveurs. Pour continuer de délivrer son service à tous ses clients avec une qualité de service acceptable, les serveurs de Znn.com, qui délivrent ordinairement des contenus multimédia, basculent en mode texte, pour délivrer des contenus uniquement textuels durant le pic d'activité. Une fois le pic de charge passé, ils peuvent revenir en mode multimédia.

L'objectif de cet exemple est d'automatiser la modification de la taille de la grappe de serveurs, ainsi que le basculement des serveurs d'un mode à un autre.

Pour cet exemple, le style architectural est défini de la manière suivante.

Au sein de la grammaire, les composants représentent les clients et des serveurs, et les connecteurs les connexions HTTP. Les clients disposent d'une propriété indiquant la latence avec laquelle ils reçoivent les informations des serveurs et une contrainte architecturale spécifie que cette valeur doit toujours être inférieure à un seuil. Le modèle de l'environnement de notre exemple est très simple : il donne la liste des serveurs disponibles qui pourront être utilisés par notre application si besoin est. De plus, il fournit la valeur de la bande passante disponible sur les différents liens de communication.

Trois opérateurs d'adaptation sont utilisables :

- *setFidelity* : basculer les serveurs du mode multimédia au mode texte, et inversement ;
- *addServer* : incrémente la taille de la grappe de serveurs ;
- *removeServer* : décrémenter la taille de la grappe de serveurs.

Les stratégies sont ici extrêmement simples, et correspondent aux quatre utilisations possibles des opérateurs (*setFidelity* permet de basculer les serveurs d'un mode à un autre).

Le vérificateur d'architecture regarde si le temps de réponse est acceptable pour tous les clients. Si la latence passe un seuil prédéfini pour un client, il déclenche le gestionnaire d'adaptation qui va déterminer s'il est nécessaire d'augmenter la taille de la grappe de serveurs ou de diminuer la qualité du contenu fourni. L'orchestrateur d'exécution de stratégie applique enfin la stratégie souhaitée.

3.5.2.4 Synthèse sur le *framework* Rainbow

Expression de contraintes de conception	Règles d'assemblage du style architectural
Réification de l'architecture d'exécution	Par le *Model Manager* implanté pour chaque système
Vérification de la conformité de l'architecture	Par l'*Architecture Evaluator* implanté pour chaque système
Mécanisme de reconfiguration	Lancement par le *Strategy Executor* de scripts Stitch [Che08] sélectionnés par l'*Adaptation Manager*
Plate-forme d'exécution	Paramétrable pour chaque système
Spécificité de l'approche	Réflexion sur la notion de style architectural qui permet de paramétrer la plate-forme Rainbow [CGS+02]

Tableau 3.1 – Résumé de l'approche Rainbow.

Le tableau 3.1 reprend les caractéristiques clé du *framework* Rainbow, selon les critères définis précédemment. Cette contribution de D. Garlan et S.-W. Cheng dispose d'une place

à part au sein des approches qui utilisent des architectures pour représenter la connaissance nécessaire au processus d'auto-adaptation. En effet, ce travail fait figure de pionnier dans le champ des systèmes auto-adaptables, et les travaux qui ont suivi y ont largement fait référence. Après avoir décrit en détails le *framework* Rainbow, nous allons maintenant nous pencher sur d'autres approches que nous évaluerons selon les mêmes critères.

3.5.3 Plastik

L'approche proposée par Plastik [GBJC07] part du constat que le processus de reconfiguration nécessite de connecter l'architecture exécutée avec une spécification d'architecture élaborée lors de la conception. Pour cela, le *framework* Plastik est architecturé autour de deux modèles : le *Runtime-Level Model*, qui représente le système exécuté, et l'*Architecture-Level Model*, qui fait référence à la connaissance de conception (figure 3.8).

D'un point de vue technologique, la plate-forme d'exécution est basée sur le modèle à composants réflexif OpenCOM [CBG+04] qui sert donc de base au *Runtime-Level Model*. L'*Architecture-Level Model* est, quant à lui, basé sur l'ADL Acme [GMW00], ainsi que sur le langage de contraintes Armani [Mon00] qui utilise des prédicats de la logique du premier ordre. Nous allons maintenant présenter l'un et l'autre.

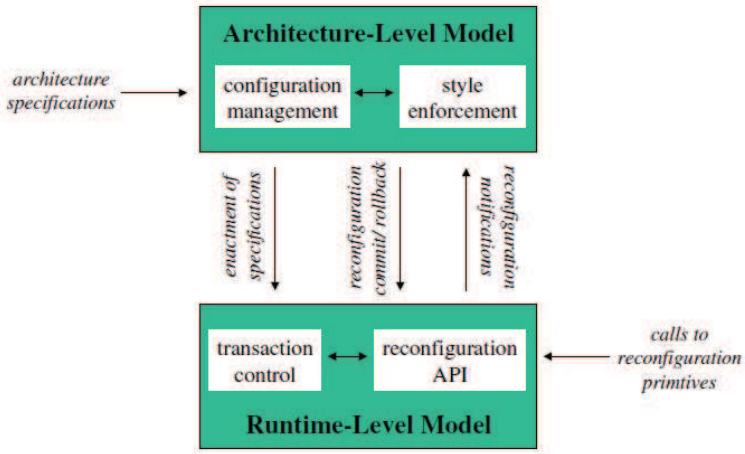

FIGURE 3.8 – Vue de haut niveau de l'approche Plastik [GBJC07].

La plate-forme d'exécution est basée sur OpenCOM, un intergiciel à composants réflexif. OpenCOM offre de nombreux points d'extension dont Plastik fait usage. En particulier, toute demande de reconfiguration de l'architecture exécutée est interceptée, pour être notifiée à la couche de plus haut niveau de Plastik qui valide ou non la modification grâce au *Style Enforcement Service*. Après validation et une fois la modification effectivement appliquée, un second événement permet de notifier la mise à jour effective de l'architecture. Notons qu'une implantation spécifique pour obtenir la quiescence et modifier l'architecture sans perte de données a été effectuée et publiée dans [PC08].

La couche de plus haut niveau est destinée à héberger les spécifications d'architecture élaborées lors de la conception, à valider les modifications souhaitées lors de l'exécution,

et à déclencher des mises à jour de la plate-forme d'exécution. Pour cela, l'architecture de conception est exprimée en utilisant l'ADL Acme, étendu afin de prendre en compte la variabilité par l'expression de morceaux d'architecture optionnels ou de dépendances entre éléments. De plus, Armani est employé afin de définir un ensemble d'invariants architecturaux. Ces différents éléments sont fournis au *Configuration management service*. Celui-ci a tout d'abord la charge d'initialiser la plate-forme à composants au lancement de l'application. De plus, à l'aide de la description architecturale exprimée dans l'ADL Acme étendu par le langage Armani, il génère un ensemble de scripts de validation et de reconfiguration pour le *Style Enforcement Service*. Lors de la détection de certains événements (exprimés dans Acme), une notification est envoyée au *Style Enforcement Service* qui peut alors valider une nouvelle architecture ou lancer des scripts de reconfiguration de la plate-forme d'exécution. Notons que les scripts sont générés dans le langage interprété Lua [IdFF96].

Le maintien du lien entre le modèle de conception et le modèle exécuté est rendu possible par la proximité entre les concepts d'Acme/Armani et d'OpenCOM, qui se basent tous les deux sur un modèle à composants. Cependant, les auteurs insistent sur la nécessité de n'utiliser qu'un sous-ensemble d'Acme pour garantir la cohérence du lien entre les deux modèles. Un tableau de correspondance entre les concepts de conception et d'exécution peut être trouvé dans [JBCG05] et [BJC05].

Les caractéristiques clé de l'approche Plastik sont résumées dans le tableau 3.2.

Expression de contraintes de conception	Par Armani [Mon00], une extension d'Acme
Réification de l'architecture d'exécution	Par OpenCOM [CBG+04]
Vérification de la conformité de l'architecture	Par script de validation généré par le *Configuration management service* pour le *Style Enforcement Service* qui effectue la validation
Mécanisme de reconfiguration	Par scripts de reconfiguration générés par le *Configuration management service* pour le *Style Enforcement Service* qui pilote ainsi OpenCOM
Plate-forme d'exécution	OpenCOM [CBG+04]
Spécificité de l'approche	Lien clairement exprimé entre architecture de conception (incluant de la variabilité) et architecture exécutée

Tableau 3.2 – Résumé de l'approche Plastik.

3.5.4 Approche de J. Kramer et J. Magee

J. Kramer et de J. Magee se sont intéressés de longue date aux ADL et sont d'ailleurs à l'origine de Darwin [MDEK95, MK96]. Ils ont fait très tôt le constat que les ADL sont très largement statiques ou ne peuvent représenter qu'un degré de variabilité peu important. Cette approche va chercher à dépasser cette limitation.

Dans celle-ci, ils se sont inspirés des architectures à trois couches présentes dans le monde de la robotique [Gat98]. Cette architecture offre l'avantage de clairement établir une séparation des préoccupations, tout en conservant des liens entre les actions de bas niveau et les buts de haut niveau. De plus, cette structuration permet une réaction immédiate pour les adaptations simples, tout en laissant la possibilité de raisonnements plus complexes dans d'autres cas.

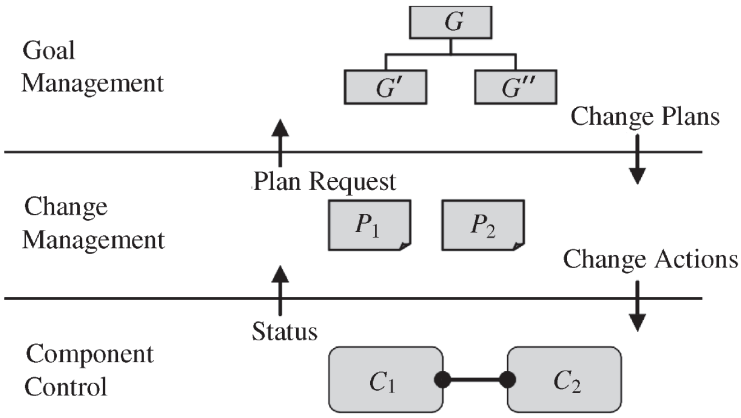

FIGURE 3.9 – Architecture à trois couches de J. Kramer et J. Magee [KM09].

Nous allons à présent détailler chacune de ces couches [SHMK08], qui sont représentées dans la figure 3.9.

Au plus bas niveau, se trouve la couche qui gère le modèle à composants qui porte l'exécution du système (*Component Control*). Cette couche maintient un modèle de l'architecture exécutée, qu'elle met à dispositions des couches supérieures. Elle prend aussi en charge la création et la suppression de composants ainsi que la modification des liens entre ceux-ci. Cette couche a enfin la responsabilité de prévenir les couches supérieures dans le cas où l'architecture exécutée se trouve dans un état invalide.

La couche du milieu (*Change management*) pilote les modifications de la couche *Component Control* à l'aide des informations fournies par celles-ci ainsi que des buts de haut niveau exprimés dans la couche *Goal Management*. La couche *Change Management* dispose d'un ensemble de plans d'action [SHMK07] qui permettent de faire évoluer l'application d'une architecture à une autre, selon un chemin prédéterminé. Dans le cas où une situation ne dispose pas de plan d'action, la couche du milieu en demande un à la couche supérieure.

La couche supérieure (*Goal Management*) produit à la demande un plan d'action à partir de l'état courant du système et des buts de haut niveau.

A présent que nous avons vu l'architecture à trois couches, nous allons considérer l'implantation. Celle-ci utilise le *Labelled Transition System Analyser* (LTSA) [KM06] pour générer une représentation LTS du modèle du domaine à partir d'un ensemble d'actions, d'un ensemble de propositions [GM03] et d'un ensemble de contraintes LTL. Les plans d'action, quant à eux, sont des sous-ensembles du modèle du domaine. Leur création est basée sur une extension de LTSA à l'aide d'algorithmes de planification [GT00, GNT04]. Une description détaillée du processus, illustrée par un exemple, peut être trouvée dans [HSMK09]. Notons enfin que la prise en compte de propriétés non-fonctionnelles a fait l'objet d'une publication spécifique [SHMK10], ainsi que l'outillage nécessaire à la gestion des modifications incrémentales de spécifications d'architectures [MMR10, MKM11].

L'exécution est basée sur Backbone [MKM06], qui fournit à la fois une plate-forme d'exécution et un ADL pour décrire des architectures. L'ADL s'appuie sur le méta-modèle des composants d'UML®2. Il permet d'exprimer des parties fixes dans l'architecture, ainsi que des parties variables. Le modèle formel de l'ADL de Backbone a été créé en utilisant le langage formel Alloy [Jac02], qui offre une combinaison de prédicats et d'algèbre relationnelle. Notons enfin qu'A. McVeigh a créé dans le cadre de sa thèse [McV09] un outil nommé Evolve [MKM11] qui permet de prévoir les évolutions d'architectures lors de la conception en utilisant une graphique qui génère une description en utilisant l'ADL de backbone.

Les caractéristiques clé de l'approche de J. Kramer et J. Magee sont résumées dans le tableau 3.3.

Expression de contraintes de conception	ADL Backbone [MKM06]
Réification de l'architecture d'exécution	Par la Couche *Component Control*
Vérification de la conformité de l'architecture	Par la Couche *Component Control*
Mécanisme de reconfiguration	Plans d'action [SHMK07] exprimés à l'aide d'extension de LTSA [KM06]
Plate-forme d'exécution	Plate-forme Backbone
Spécificité de l'approche	Réflexion sur les ADL pour inclure de la variabilité. Architecture à trois couches issue de la robotique

Tableau 3.3 – Résumé de l'approche de J. Kramer et J. Magee.

3.5.5 Prism-MW et DIF

L'approche proposée dans [MMMR12] s'intéresse aux applications distribuées. En effet, avec celles-ci, il peut exister un nombre très important d'architectures possibles pour fournir une même fonctionnalité, avec des niveaux de qualité de services pouvant varier fortement de l'une à l'autre. Par exemple, la latence d'un système peut être améliorée si les échanges de données les plus fréquents et volumineux sont réalisés au sein du même nœud ou avec des liaisons fiables et rapides.

Pour résoudre ce point dur, les auteurs ont repris l'architecture à trois couches présentée dans le paragraphe précédent. Dans ce cadre, ils ont conçu un *framework* à composants nommé Prism-MW [MMRM05] pour prendre en charge les deux couches basses. Ils ont de plus réalisé une couche de gestion des objectifs (*Goal Management*) nommée DIF (*Deployment Impropvement Framework*) qui a pour but de permettre un déploiement optimal d'une application sur plusieurs nœuds. De plus, DIF peut réagir aux modifications de l'environnement, pour effectuer un redéploiement afin d'optimiser le placement des composants au nouveau contexte d'exécution. L'architecture générale du système est présentée sur la figure 3.10.

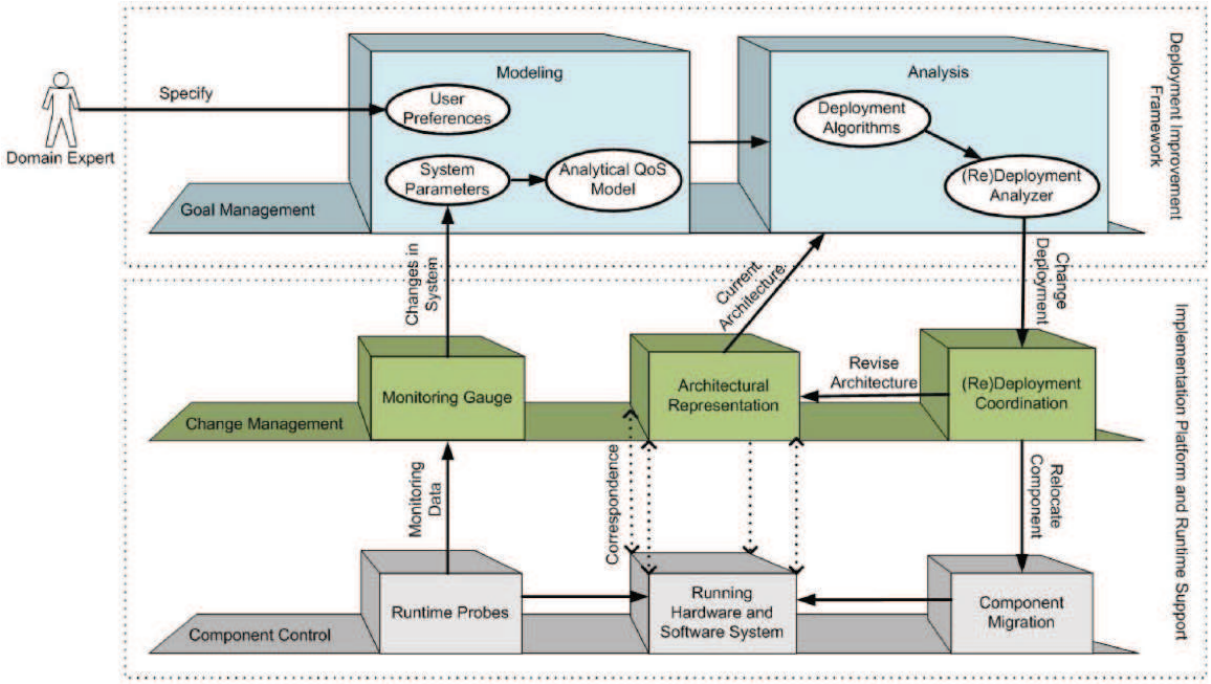

FIGURE 3.10 – Architecture à trois couches, revisitée par [MMMR12].

Nous allons à présent donner davantage de détails à propos de Prism-MW, puis nous étudierons ensuite de manière plus approfondie DIF. Prism-MW dispose de quatre fonctionnalités de base permettant de modifier l'architecture exécutée :

- **la migration de composants** permet de déplacer un composant logiciel vers un hôte (ordinateur, machine virtuelle...) ;
- **la coordination de (re)déploiement** permet d'effectuer les opérations de création ou de modification de composant dans un ordre cohérent, tout en préservant les états ;
- **l'analyse de l'exécution** permet de collecter des données de bas niveau qui qualifient l'état courant de la plate-forme d'exécution ;
- **l'agrégation de données dans des jauges** permet de fournir des mesures de plus haut niveau à DIF, en corrélant et agrégeant des données de bas niveau.

La couche DIF prend en charge deux activités :

- **l'activité de modélisation :** un modèle architectural extensible permet l'ajout de paramètres système, de variables évaluant la qualité de service à l'aide de ces paramètres et de spécifications d'objectifs utilisateur en termes de qualité de service. Ce modèle permet d'évaluer toute configuration de (re)déploiement à l'aide d'une fonction d'utilité.
- **l'activité d'analyse :** un ensemble d'algorithmes parcourent l'ensemble des configurations possibles de (re)déploiement et évaluent les conséquences des modifications sur les paramètres du système (et donc sur sa qualité de service) ainsi que sur son temps de reconfiguration.

La conception du modèle architectural pris en charge par l'activité de modélisation a fait l'objet d'une attention particulière de la part des auteurs. Constatant que des langages tels qu'UML® ou AADL ne permettaient pas de représenter tous les concepts souhaités, un modèle spécifique a été créé et implanté en utilisant xADL et FSP [EMM07, EM08]. Ce modèle permet de définir une topologie distribuée, d'ajouter des propriétés sur chaque élément (nœud, lien, service...), de définir des contraintes, des indicateurs de qualité de service et des fonctions d'utilité pour capturer les préférences des utilisateurs en termes de qualité de service [MMMR12].

Pour chaque application, DIF doit donc être configuré pour disposer des paramètres système à prendre en compte et des variables de qualité de service à considérer ainsi que les objectifs à atteindre. DIF se met alors en lien avec Prism-MW, qui lui fournit les données relatives à l'exécution. A l'aide de celles-ci, DIF peut calculer la valeur issue de la fonction d'utilité pour la configuration actuelle et pour un ensemble de configurations envisagées, déclenchant ainsi un (re)déploiement si un nouveau plan de déploiement optimal est trouvé.

Expression de contraintes de conception	Dans le modèle architectural de DIF
Réification de l'architecture d'exécution	Prise en charge par le cœur de Prism-MW
Vérification de la conformité de l'architecture	Focus de l'approche davantage sur l'optimisation (prise en charge par DIF) que sur la validité
Mécanisme de reconfiguration	Déclenché par l'évaluation de la fonction d'utilité qui indique une nouvelle topologie que Prism-MW doit déployer
Plate-forme d'exécution	Prism-MW [MMRM05]
Spécificité de l'approche	Considère l'optimisation de systèmes distribués en utilisant une architecture trois couches

Tableau 3.4 – Résumé de l'approche Prism-MW et DIF.

Récemment, ces travaux ont été prolongés en remplaçant DIF par FUSION (*FeatUre-oriented Self-adaptatION*) [EEM13], une nouvelle couche de *Goal Management* qui s'appuie elle aussi sur Prism-MW et dont l'ambition est double :

- apporter une meilleure structuration des possibilités d'adaptation par une approche rejoignant les lignes de produits dynamiques, avec un diagramme de fonctionnalités ;
- utiliser une base de connaissances distante, qui offre des fonctionnalités d'auto-apprentissage pour améliorer les adaptations les unes après les autres.

Les caractéristiques clé de l'approche Prism-MW et DIF sont résumées dans le tableau 3.4.

3.5.6 Architectural Runtime Configuration Management (ARCM)

L'approche ARCM (*Architectural Runtime Configuration Management*) [GvdHT05, DAH+07, GvdHT09] a été développée par J. C. Georgas dans le cadre de sa thèse encadrée par R. N. Taylor et en collaboration avec A. van der Hoek. Au centre de cette approche se trouve un modèle architectural du système exécuté qui représente la configuration de celui-ci à chaque instant. Les configurations successives présentent l'originalité d'être organisées dans un graphe qui présente un point de vue historisé du processus d'adaptation en permettant de parcourir les configurations adoptées successivement par le système. De plus, ce modèle dispose d'un ensemble de métadonnées relatives au processus d'adaptation, qui indiquent, par exemple, le nombre de fois que le système s'est trouvé dans une configuration donnée et le temps qu'il a maintenu celle-ci.

ARCM offre aussi la possibilité de naviguer dans l'historique des configurations en annulant des adaptations effectuées automatiquement ou en forçant le processus d'adaptation dans une configuration préenregistrée. De plus, l'historique des configurations permet aux administrateurs de mieux comprendre le comportement dynamique du système. Les concepteurs, quant à eux, peuvent s'appuyer sur cette connaissance pour améliorer le processus d'adaptation.

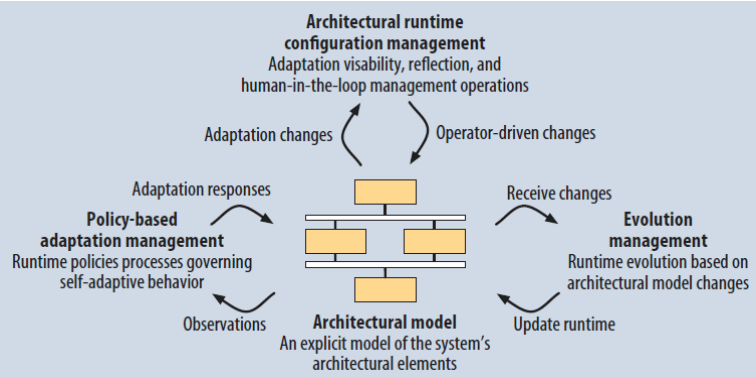

FIGURE 3.11 – Vue de haut niveau de l'approche ARCM [GvdHT09].

La figure 3.11 montre une vue de haut niveau de l'approche ARCM. En son centre, se trouve le modèle du système. Autour de celui-ci, un ensemble de processus gèrent l'évolution, l'adaptation et les configurations que peut adopter le système.

Les outils de gestion de l'évolution (*Evolution management*) sont notifiés en cas de mise à jour du modèle architectural. Ils ont alors pour rôle de mettre à jour le système administré afin que celui-ci corresponde à ce qui a été décidé au niveau modèle.

Les outils de gestion de l'adaptation (*Adaptation management*) déterminent les changements à appliquer au modèle architectural. Pour cela, la logique d'adaptation est distribuée dans un ensemble de politiques d'adaptation indépendantes les unes des autres. Les politiques sont exécutées en fonction de pré-conditions et agissent au niveau des composants architecturaux en provoquant leur ajout, leur retrait ou en modifiant les liaisons entre ceux-ci. Notons que les politiques peuvent être ajoutées ou supprimées à la volée lors de l'exécution.

Enfin, un dernier processus gère dans un graphe l'historisation des configurations successives adoptées par le système. Chaque nœud du graphe représente une configuration. Deux nœuds sont donc annotés de manière spécifique : celui qui représente la configuration initiale et celui qui représente la configuration actuelle. Entre les nœuds, les arcs représentent les transitions effectuées par le système. Sur chacune d'elle, un calcul de différence entre la configuration source et la configuration cible est indiqué. Une interface graphique a été développée. Celle-ci, en plus d'opérations de visualisation du graphe, offre la possibilité à un administrateur de forcer le système à adopter une configuration particulière ou, au contraire, d'annuler une modification effectuée de manière automatique.

L'implantation repose sur ArchStudio [DvdHT02]. Cet IDE basé sur Eclipse a été développé par E. M. Dashofy dans le cadre de sa thèse encadrée par R. N. Taylor [Das07]. Cet atelier extensible propose une interface pour modéliser, visualiser et analyser et corriger des architectures en utilisant un DSL basé sur XML : xADL 2.0 [DvdHT05].

Les caractéristiques clé de l'approche ARCM sont résumées dans le tableau 3.5.

Expression de contraintes de conception	Fonctionnalité souhaitée, mais non implantée [GT08]
Réification de l'architecture d'exécution	Au cœur d'ARCM, en utilisant la formalisation de xADL 2.0
Vérification de la conformité de l'architecture	Non implantée
Mécanisme de reconfiguration	Basé sur des politiques, qui sont des tuples événement / condition /action
Plate-forme d'exécution	Gérée et réifiée par ArchStudio
Spécificité de l'approche	Gestion d'un graphe qui présente les modèles correspondants aux états successifs du système

Tableau 3.5 – Résumé de l'approche ARCM.

3.5.7 Synthèse

La question de la gestion de la connaissance est centrale dans l'informatique autonomique et plus largement dans le cadre des systèmes auto-adaptables. Parmi cette connaissance, les modèles architecturaux sont apparus rapidement comme devant jouer un rôle de premier plan [OMT98, OGT+99]. En effet, un modèle architectural qui évolue simultanément au système administré pour en refléter l'état, offre de multiples avantages [GCH+04, GSC09]. Tout d'abord, en tant que modèle abstrait, il fournit une perspective globale sur le système et permet de mettre en avant les propriétés et comportements importants du système. De plus, il permet de rendre explicites les contraintes d'intégrité d'un système ; cela peut permettre d'établir le périmètre des modifications autorisées et de s'assurer de la validité d'un changement. Enfin, un modèle peut être associé avec des contraintes architecturales qui, par exemple, interdisent des cycles dans l'architecture.

A partir du début des années 2000, de nombreux travaux se sont déployés dans cette direction. Rainbow fait parmi ceux-ci figure de pionnier. Cette approche a proposé un découpage strict de la logique d'adaptation pour permettre une meilleure séparation des préoccupations. De plus, un travail important a été mené sur la notion de style architectural afin de paramétrer la plate-forme Rainbow, lui permettant ainsi d'être adaptable pour une vaste gamme de systèmes.

Par la suite, les travaux successifs ont cherché à approfondir des points différents. Plastik s'est penché sur le lien entre l'architecture de conception et le modèle de l'architecture exécuté. Des travaux de J. Kramer et J. Magee ont, quant à eux, investigué la variabilité au sein des ADL, tout en proposant une architecture d'exécution à trois couches, inspirée des travaux en robotique. Les travaux de Prism-MW et DIF ont repris cette architecture à trois couches, en l'appliquant aux systèmes distribués. Enfin, l'approche ARCM a proposé un graphe qui permet de naviguer dans un historique des configurations adoptées successivement par un système au cours de son exécution.

Pour terminer, nous pouvons aussi élargir notre perspective en soulignant que si la gestion de l'aspect structurel des applications (en ayant recours aux architectures) a montré des résultats très prometteurs, d'autres approches moins étudiées considèrent des angles de vue complémentaires. C'est le cas par exemple de l'étude de la dynamique comportementale des programmes, qui reste encore peu étudiée [ZC06].

3.6 Conclusion

L'informatique autonomique a pour but l'autogestion (notament l'auto-configuration, l'auto-optimisation, l'auto-réparation et l'auto-protection) des systèmes à partir d'objectifs de haut niveau fournis par les administrateurs. Pour bâtir ces systèmes autogérés, l'informatique autonomique s'est inspirée de nombreux domaines et, notament, de la théorie du contrôle. Cette influence clé a permis à IBM de proposer une architecture de référence [IBM06] largement connue aujourd'hui.

Celle-ci repose sur un ou plusieurs éléments autonomiques en relation. Chacun d'eux est composé d'un élément administré et d'un gestionnaire autonomique, dissociant ainsi le code applicatif du code d'adaptation. Le gestionnaire autonomique a la charge de surveiller un système via les capteurs, d'analyser son état pour le corriger si besoin à l'aide d'une série d'actions transmises à l'élément administré par les actionneurs.

En raison de la complexité d'implantation des gestionnaires autonomiques, IBM a proposé une architecture appelée MAPE-K qui sépare les préoccupations à l'aide de quatre fonctions qui partagent une base de connaissances :

- **la fonction de supervision** collecte, agrège et filtre les données en provenance de l'élément administré via ses capteurs ;
- **la fonction d'analyse** modélise la situation pour détecter la présence de problèmes ;
- **la fonction de planification** utilise la liste de problèmes transmise par la fonction d'analyse pour définir un plan d'action afin de corriger l'état de l'élément administré ;
- **la fonction d'exécution** exécute un plan d'actions défini par la fonction planification ;
- **la base de connaissances** est partagée par les quatre fonctions ci-dessus.

L'un des points clé de l'architecture MAPE-K est la connaissance. Grâce à celle-ci, le système peut être analysé plus finement en donnant, par exemple, davantage de sens aux informations remontées par la fonction de supervision. De plus, la connaissance peut aussi permettre à la fonction de planification d'utiliser les résultats de précédentes adaptations.

Parmi les structurations possibles de cette connaissance, l'une d'entre elles nous apparaît particulièrement intéressante. Il s'agit de la modélisation de l'architecture exécutée à partir des données remontées au gestionnaire autonomique. Cela permet de disposer des avantages liés aux architectures et, notamment, un point de vue de haut niveau sur l'ensemble du système. Nous avons donc étudié un assemble d'approches qui s'appuient sur des architectures, en commençant par Rainbow qui fait figure de pionnier dans ce domaine.

Cette étude nous a permis de montrer que les différentes approches déléguent la gestion du ou des modèles au développeur, et ne donnent que peu d'indications quant à l'implantation concrète de ceux-ci. Pour cette raison, il nous semble important d'investiguer cela. Ce sera l'un des objets de notre proposition.

Deuxième partie

Contribution

Chapitre 4

Proposition

Sommaire

4.1 Problématique et objectifs **98**
4.2 Approche **100**
4.2.1 Présentation générale 100
4.2.2 Formalisation des architectures 101
4.2.3 Vérification de la validité entre modèles 103
4.2.4 Synthèse 105
4.3 Etude de la variabilité dans les différentes architectures **106**
4.3.1 Variabilité pour l'architecture de conception 106
4.3.2 Variabilité pour l'architecture de déploiement 108
4.3.3 Synthèse 108
4.4 Conclusion **109**

4.1 Problématique et objectifs

Dans la première partie de ce manuscrit, nous avons présenté la notion d'adaptation logicielle. Nous nous sommes en particulier attardés sur l'informatique orientée service qui est fortement utilisée aujourd'hui pour permettre des adaptations à l'exécution. Cette nouvelle approche est basée sur les principes de faible couplage, de liaison retardée et de substituabilité de modules logiciels appelés service. L'informatique orientée service permet ainsi d'aborder de nouveaux domaines d'application caractérisés notamment par de fortes contraintes de dynamisme et par une faible prédictibilité. Nous avons également montré que, de façon non surprenante au vu de ses propriétés, l'informatique orientée service rencontre un réel succès pour la mise en place d'applications ambiantes (ou pervasives). Ces applications sont aujourd'hui de plus en plus répandues.

Dans une seconde partie, nous avons présenté la notion d'informatique autonomique. Nous pensons en effet que des propriétés autonomiques telles que l'auto-configuration, l'auto-réparation, l'auto-optimisation ou encore l'auto-protection peuvent permettre de faire face à la complexité croissante des besoins en adaptation. Ces propriétés peuvent permettre de diminuer significativement les coûts d'exploitation et de maintenance d'une part et palier l'absence d'administrateurs d'autre part.

Voici une vision synthétique des constatations que nous avons faites le long de ces chapitres dédiés à l'état de l'art :

- **les besoins en adaptation vont croissant dans les applications modernes.** L'évolution fréquente des besoins, les changements de disponibilité des ressources, la nécessité de corriger des dysfonctionnements et la prise en compte de la mobilité des utilisateurs nécessitent des interventions complexes et fréquentes sur les éléments logiciels.
- **l'interruption de service est aujourd'hui mal acceptée.** Le modèle économique des entreprises nécessite un fonctionnement en continu (99% de disponibilité correspond déjà à une indisponibilité de plus de 3 jours et demi sur une année). Les activités telles que le e-commerce ou le e-manufacturing ne permettent pas les arrêts des fonctions.
- **l'informatique orientée service est très utilisée aujourd'hui.** Cette approche a été adoptée dans de nombreux domaines tels que la maison connectée, la ville intelligente ou l'industrie manufacturière.
- **l'approche orientée service permet une grande souplesse mais soulève des problèmes de suivi.** Dès lors que le système décide lui-même de certaines évolutions, il devient difficile pour les administrateurs de comprendre les adaptations et d'intervenir efficacement en cas de dysfonctionnement.
- **l'informatique autonomique représente un axe d'évolution majeur.** La complexité des systèmes ne baissera pas et il devient impératif de trouver des solutions d'administration automatisées ne requérant pas d'arrêt. L'approche autonomique promeut en particulier l'auto-gestion et permet d'envisager des administrations plus pérennes et moins couteuses des systèmes.

- **l'informatique autonomique repose sur la notion de connaissance.** Aujourd'hui cette connaissance est souvent peu formalisée, relativement ad hoc et peu généralisable.

Notre objectif est de mieux formaliser la connaissance dans le cadre des systèmes orientés services et, notamment, pour les applications pervasives. Ceci dans le but de faciliter le travail des administrateurs ou la construction de gestionnaires autonomiques. Nous souhaitons nous focaliser sur le niveau architectural qui nous semble être le bon niveau de granularité pour mener à bien des adaptations. Comme nous l'avons vu dans l'état de l'art, disposer d'une formalisation claire de l'architecture exécutée est très précieux pour guider les adaptations. Cependant, cette architecture seule n'est pas suffisante. En effet, il est nécessaire de disposer d'une autre connaissance architecturale qui donne l'ensemble de propriétés que l'on souhaite garantir à l'exécution et qui exprime ce qu'est une architecture valide afin de conduire les adaptations conformément à cette description. Cette seconde moitié de connaissance qui est issue de la phase de conception, sera nommée par la suite architecture de conception.

Cette complémentarité entre l'architecture de l'exécution et l'architecture de conception avait déjà été perçue par Schmerl et Garlan [SG02]. Pour eux, l'utilisation d'outils architecturaux lors de la conception et de l'exécution est pertinente ; car elle permet :

- de réutiliser la connaissance acquise lors de la phase de conception pour guider l'adaptation ;
- d'offrir une continuité de point de vue entre la conception et l'exécution ;
- de réutiliser à l'exécution des analyses issues de la conception afin de mieux comprendre les conséquences des changements à l'exécution.

Nous souscrivons à cette analyse, mais nous constatons cependant qu'architecture de conception et architecture de l'exécution sont à la fois analogues (ce sont des architectures) et différentes (par exemple, l'architecture de conception peut posséder de la variabilité, contrairement à l'architecture de l'exécution). Notre approche va reposer sur l'investigation de ces complémentarités et différences, afin de structurer la connaissance des gestionnaires autonomiques. Plus précisément, nos objectifs sont de :

- formaliser la notion d'architecture de conception ;
- formaliser la notion d'architecture de l'exécution ;
- formaliser les liens entre ces deux types d'architecture ;
- proposer une approche algorithmique pour calculer les liens entre ces deux types d'architecture durant l'exécution ;
- proposer un atelier mettant en évidence ces différents éléments de façon à aider les administrateurs et à constituer une base de connaissances pour un éventuel gestionnaire autonomique ;
- appliquer nos travaux au domaine de l'informatique pervasive.

4.2 Approche

4.2.1 Présentation générale

Notre objectif est de simplifier les tâches des administrateurs d'applications à base de composants orientés services, en maintenant des liens de traçabilité entre les architectures réalisées à différents moments du cycle de vie de l'application [GHL13a, GHL13b, LCGH14]. En effet, comme nous l'avons vu précédemment, l'architecture de l'exécution évolue suite à des modifications de l'environnement d'exécution (arrivée et/ou départ d'équipements). L'administrateur doit être capable de savoir si l'exécution qui est en cours est toujours valide par rapport à celle qui est souhaitée. Nous souhaitons automatiser la vérification de la validité de l'architecture à l'exécution. Pour ce faire, nous définissons trois types d'architectures, qui sont utilisées à différentes étapes du cycle de vie :

- **l'architecture de conception** est définie par les architectes du domaine. C'est une composition de différents éléments liés les uns aux autres. Dans le cadre de cette thèse, nous travaillons avec des architectures de conception qui comprennent de la variabilité, comme présentée dans le chapitre 3 page 59. Ces architectures seront raffinées dans les étapes suivantes du processus de développement pour en réduire la variabilité.
- **l'architecture de déploiement** est issue de l'architecture précédente et elle est destinée à une plate-forme d'exécution donnée. Elle décrit le système qui devra être instancié ainsi que les artéfacts nécessaires pour cela. La configuration initiale des composants est fournie. Notons que cette architecture présente uniquement la variabilité que la plate-forme sera capable de résoudre (comme, par exemple, la résolution des dépendances de services).
- **l'architecture de l'exécution** est la représentation de l'application en exécution. Cette architecture est construite grâce à l'analyse de l'exécution avec des mécanismes de supervision. Elle est une vue de l'exécution avec un certain niveau d'abstraction : toutes les informations de l'exécution ne sont pas représentées. Dans cette architecture, il n'existe plus de variabilités ; ces dernières ont toutes été résolues par la machine d'exécution. Notons que l'architecture de l'éxécution peut évoluer très fréquement.

Nous souhaitons mettre en place une approche automatisée afin de conserver les liens entre les différentes architectures au fur et à mesure de leurs évolutions. Pour cela, nous proposons de recourrir à l'informatique autonomique. La figure 4.1 illustre notre approche. Nous ajoutons « au-dessus » de notre système une boucle autonomique MAPE-K [KC03] et nous nous intéressons plus particulièrement à la structuration de sa connaissance. C'est dans la connaissance que nous voulons maintenir le lien entre les architectures de conception et de l'exécution, calculé lors de la phase d'analyse. Notons que l'architecture de déploiement n'apparaît pas dans cette figure. En effet, elle n'est utile que pour le premier déploiement de l'application sur la plate-forme. Sitôt instancié, le système est réifié sous la forme d'une architecture de l'exécution.

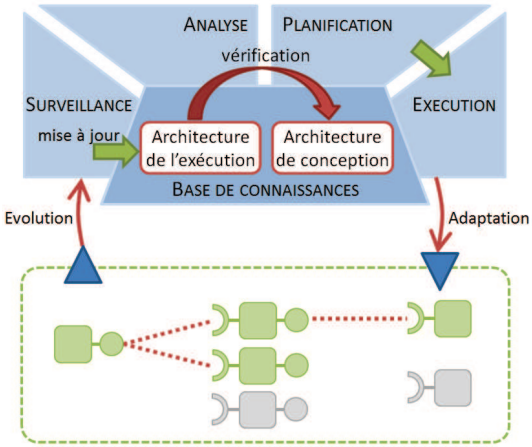

FIGURE 4.1 – Principes de notre approche.

Cette approche requiert de mettre en place des capteurs et des actionneurs sur le système permettant, à la phase de supervision, de mettre à jour l'architecture de l'exécution et, à la phase d'exécution, d'adapter le système selon ce qui a été planifié. Pour cette partie, nous nous basons sur les travaux déjà réalisés dans la thèse de D. Morand [Mor13].

Dans les deux parties suivantes, nous allons présenter la formalisation de la connaissance, puis la méthode de vérification de la validité entre les architectures.

4.2.2 Formalisation des architectures

La formalisation des architectures a pour objectif d'identifier clairement les différents éléments qui les composent. Pour ce faire, notre proposition est basée sur la définition d'un ensemble de méta-modèles [OMG02, Fav04]. Un méta-modèle permet de spécifier de manière claire, précise et non-ambiguë le langage utilisé par un modèle. Plus précisément, un méta-modèle sert à définir une grammaire et un vocabulaire afin de réaliser des modèles conformes et cohérents à celui-ci. Dans notre cas, nous proposons de définir un méta-modèle pour chaque type d'architecture afin de pouvoir construire des architectures conformes à celui-ci (figure 4.2).

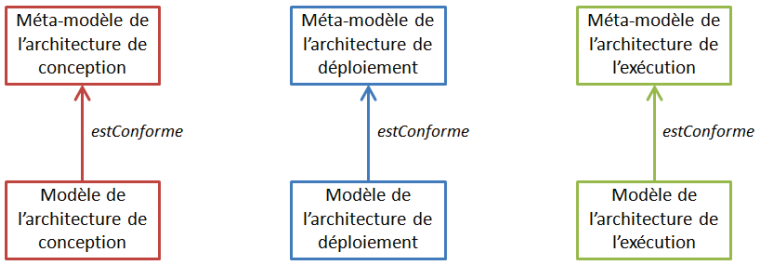

FIGURE 4.2 – Conformité entre les modèles et les méta-modèles.

La définition de ces trois méta-modèles met en œuvre un ensemble de concepts communs ; car tous les trois *sont* des méta-modèles architecturaux (même si chacun dispose de spécificités propres à son domaine d'application). Nous proposons donc de concevoir un méta-modèle commun aux trois méta-modèles d'architectures. Celui-ci sera hérité par ces derniers, comme illustré à la figure 4.3.

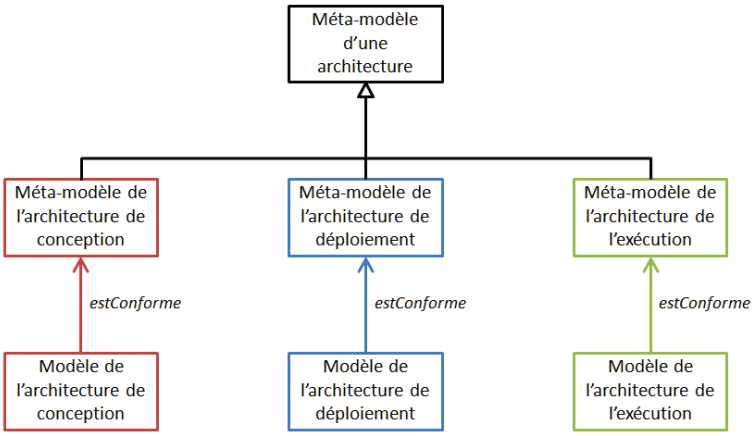

FIGURE 4.3 – Organisation générale des méta-modèles.

Cette organisation permet de mettre en avant la proximité entre les différentes architectures et de favoriser ainsi le passage de l'une à l'autre. De plus, nous établirons par la suite des liens entre les composants des différentes architectures. En effet, nous désirons avoir la même continuité de point de vue entre les composants que celle que nous souhaitons entre les architectures.

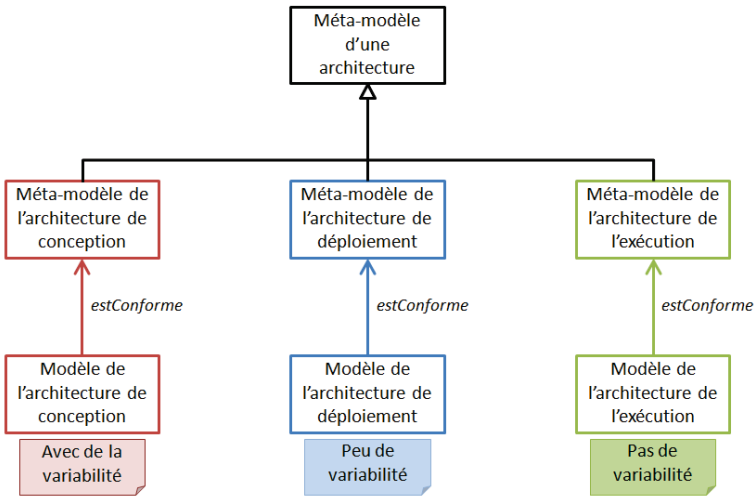

FIGURE 4.4 – Niveaux de variabilité dans les différentes architectures.

Comme nous l'avons présenté en introduction de cette partie, les différentes architectures sont définies avec des niveaux de variabilité qui diffèrent (figure 4.4). L'architecture

de conception est la plus variable, comme dans le cadre des lignes de produits logicielles [KCH+90]. Le méta-modèle doit donc supporter l'expression de cette variabilité. L'architecture de déploiement est un raffinement d'une architecture de conception ; elle contient moins de variabilité mais respecte les contraintes de l'architecture de conception dont elle est issue. Les seuls éléments de variabilité qui restent sont ceux qui pourront être résolus par la machine d'exécution. L'architecture de l'exécution est une vue de ce qui s'exécute et, par conséquent, il n'y a plus aucune variabilité dans ce modèle.

Dans le chapitre suivant, nous allons détailler l'ensemble des méta-modèles qui ont été présentés dans cette partie.

4.2.3 Vérification de la validité entre modèles

Nous venons de présenter les différents modèles et méta-modèles de notre approche. Nous allons maintenant nous pencher sur les liens qui unissent ceux-ci. Plus précisément, nous allons décrire comment la validité d'une architecture de l'exécution peut être vérifiée par rapport à son architecture de conception. Pour cela, deux mécanismes sont mis en place pour prendre en compte des moment distincts du cycle de vie :

- lors de la conception et du déploiement de l'application : par construction ;
- lors de l'évolution du système pendant son exécution : avec un algorithme de validation.

Tout d'abord, la validité de l'architecture exécutée suite à son déploiement peut être vérifiée par construction : l'architecture de déploiement est issue de l'architecture de conception, en restreignant la variabilité de cette dernière. De manière analogue, l'architecture de l'exécution est issue de l'architecture de déploiement au démarrage. Cela peut être modélisé par des relations de transformations entre les méta-modèles (même si nous ne les avons pas outillées dans ce travail de thèse), comme illustré dans la figure 4.5.

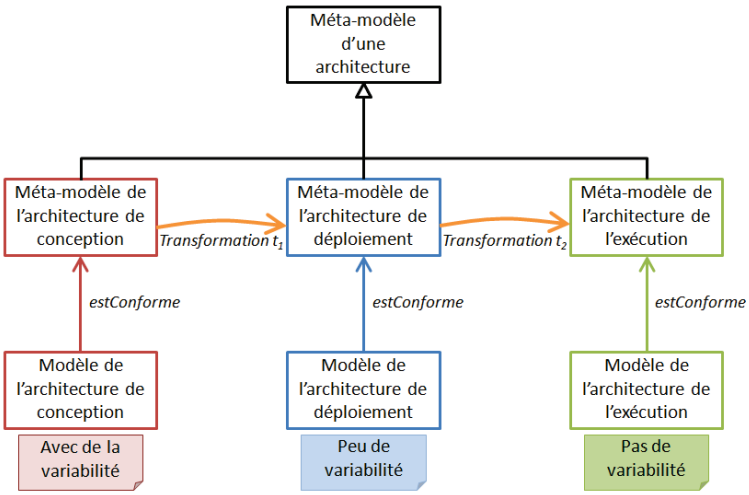

FIGURE 4.5 – Les transformations entre méta-modèles.

Ces deux transformations ne sont pas du même type. La première transformation entre le méta-modèle de l'architecture de conception et celui de l'architecture de déploiement est l'expression de la spécialisation/configuration d'un certain nombre d'éléments. La deuxième transformation entre le méta-modèle de l'architecture de déploiement et celui de l'architecture de l'exécution est la projection sur une plate-forme d'exécution. Evidemment, ces transformations exprimées au niveau méta-modèle doivent être valides au niveau modèle.

Pendant l'exécution, l'architecture de l'exécution évolue à cause de changements dans l'environnement d'exécution. Nous proposons de mettre en œuvre un algorithme de validation qui s'assure que les éléments modifiés à l'exécution respectent les contraintes définies dans l'architecture de conception (figure 4.6). La difficulté principale pour cette vérification est de savoir à quel moment elle doit être opérée. Il faut que la vérification se fasse lors d'un état « stable » et non dans un état où des modifications « en cascade » ne sont pas terminées. De plus, l'algorithme de vérification surcharge le fonctionnement dit normal de l'application, il ne faut pas l'appeler trop fréquemment, ce qui est un risque lors de modifications très fréquentes de l'architecture exécutée (par exemple, quand un équipement est en limite de portée et ne peut être détecté en continu).

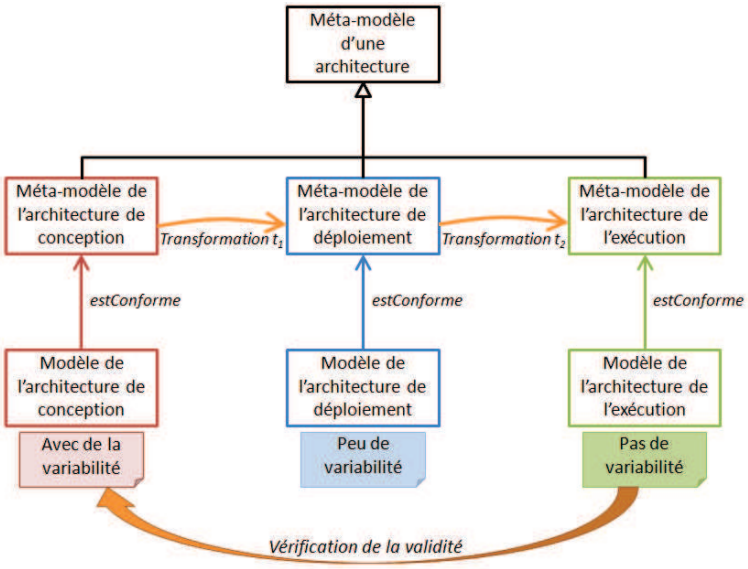

FIGURE 4.6 – Vérification de la validité de l'exécution par rapport à la conception.

Dans le chapitre suivant, nous détaillerons les différents méta-modèles ainsi que l'algorithme de vérification de la conformité entre l'architecture de l'exécution et celle de la conception.

4.2.4 Synthèse

Notre proposition repose sur trois types d'architectures (tableau 4.1) qui sont créées et mises à jour durant le cycle de vie de l'application. L'architecture de conception, comme son nom l'indique sera définie lors de l'étape de conception. Bien entendu, cela n'exclut pas que celle-ci puisse être mise à jour lors de l'exécution de l'application, même si ce n'est pas sa vocation première. L'architecture de déploiement est, elle aussi, définie à la conception. Elle est ensuite fournie à la plate-forme d'exécution pour initialiser l'application au début de l'étape d'exécution. Notons que cette architecture n'est plus utilisée une fois l'application démarrée. Enfin, l'architecture de l'exécution est créée et maintenue à jour uniquement lors de la phase d'exécution.

	Architecture de conception	**Architecture de déploiement**	**Architecture de l'exécution**
Moment de définition dans le cycle de vie	conception	conception	exécution
Moment d'utilisation dans le cycle de vie	conception et exécution	initialisation de l'exécution	exécution
Variabilité	oui	uniquement celle que la plate-forme d'exécution peut traiter	non

Tableau 4.1 – Comparaison des différentes architectures.

Une architecture clé de notre approche est donc l'architecture de conception. Celle-ci intègre de la variabilité, telle que l'on peut en trouver dans les lignes de produits dynamiques (section 2.5 page 46). La variabilité peut être exprimée dans l'architecture de conception dans la mesure où celle-ci permet de représenter l'ensemble des états valides du système lors de l'exécution. A l'opposé, la variabilité est totalement absente de l'architecture de l'exécution qui rend compte de ce qui est à un instant donné. Entre les deux, l'architecture de déploiement peut présenter de la variabilité, à condition que celle-ci puisse être prise en compte par la plate-forme d'exécution lors de l'initialisation de l'application.

Lors de l'exécution, il est important de pouvoir vérifier que l'architecture de l'exécution est valide par rapport à son architecture de conception. Pour cela, notre contribution s'attachera à caractériser la vérification de la validité et à donner les algorithmes pour son implantation.

4.3 Etude de la variabilité dans les différentes architectures

Nous avons expliqué précédemment qu'il est possible d'introduire de la variabilité dans les architectures de conception et de déploiement. Les points de variabilité doivent être clairement spécifiés pour que nous puissions mettre en place l'algorithme de vérification de l'architecture de l'exécution par rapport à l'architecture de conception. Dans cette section, nous allons présenter les différents types de variabilité de notre approche.

4.3.1 Variabilité pour l'architecture de conception

L'architecture de conception est constituée d'un ensemble de composants mis en relation, qui permettent de définir les états valides que le système pourra prendre lors de son exécution, de manière analogue à ce que nous avons présenté dans le cadre des lignes de produits (section 2.4 page 37). L'architecture de conception intègre des points de variabilité au niveau des liaisons et des composants.

4.3.1.1 Variabilité topologique au niveau des liaisons

La variabilité peut être structurelle ; c'est-à-dire se situer au niveau de l'assemblage des composants. Pour cela, nous avons défini des cardinalités sur chacune des extrémités des liens entre ces composants.

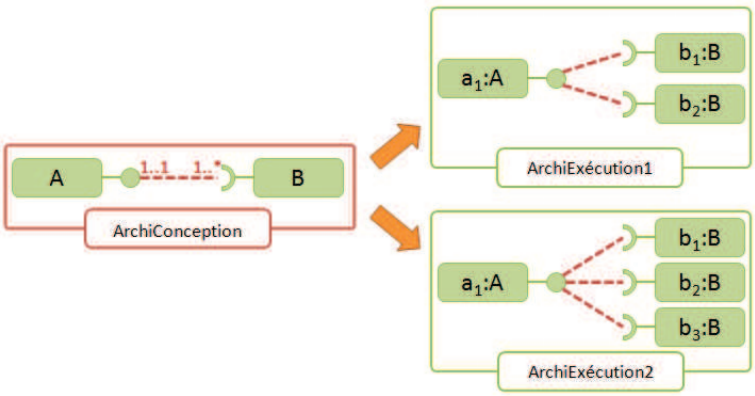

FIGURE 4.7 – Illustration de la variabilité topologique.

Par exemple, la figure 4.7 présente deux architectures exécutées possibles (*ArchiExécution1* et *ArchiExécution2*) qui sont valides par rapport à l'architecture de conception (*ArchiConception*). L'architecture de conception comprend deux composants (*A* et *B*) en relation par une liaison. Cette liaison est annotée par des cardinalités qui offrent la possibilité de lier, lors de l'exécution, plusieurs instances de *B* à une instance de *A*. Les architectures proposées (*ArchiExécution1* et *ArchiExecution2*) respectent les contraintes définies dans l'architecture de conception.

4.3.1.2 Variabilité au niveau d'un composant

Un second niveau de variabilité permet, lors de l'exécution, la substitution d'une instance de composant par une autre, alors que leurs implantations sont différentes. Cette substitution est possible grâce à la définition d'une spécification du composant souhaité au niveau de l'architecture de conception. Une spécification définit au minimum un ensemble d'interfaces fournies et requises. Ainsi, toute implantation de composant qui implante une spécification, doit disposer des interfaces de cette dernière.

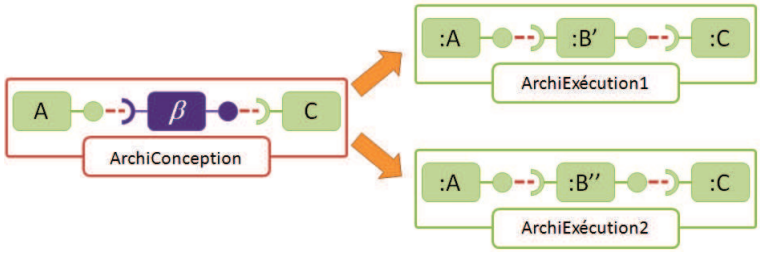

FIGURE 4.8 – Illustration de la variabilité au niveau d'un composant.

Ce mécanisme offert par les spécifications de composants est illustré par la figure 4.8. Celle-ci présente deux architectures de l'exécution *ArchiExécution1* et *ArchiExécution2* qui sont valides par rapport à une même architecture de conception *ArchiConception*. Cette dernière contient β, une spécification de composant. En supposant que les implantations de composants B' et B'' implantent la spécification β, les architectures de l'exécution *ArchiExécution1* et *ArchiExécution2* sont valides par rapport à leur architecture de conception *ArchiConception*.

4.3.1.3 Variabilité au niveau de la configuration d'un composant

La variabilité peut aussi se situer au niveau de la configuration des composants en leur adjoignant des contraintes. Celles-ci définissent, par exemple, des plages de valeurs possibles pour certains paramètres lors de la phase de conception. A l'exécution, ces paramètres doivent avoir une valeur valide par rapport à ce qui a été défini précédemment.

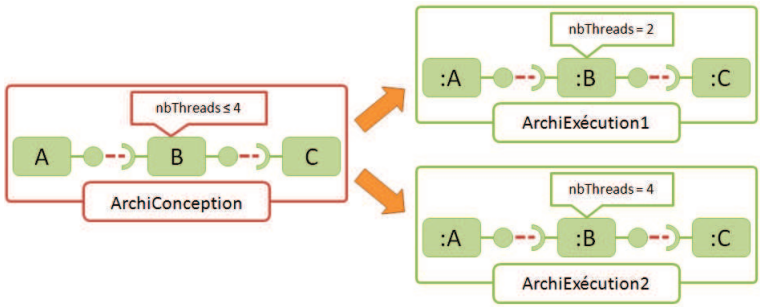

FIGURE 4.9 – Illustration de la variabilité au niveau de la configuration d'un composant.

La figure 4.9 présente deux architectures *ArchiExécution1* et *ArchiExécution2* qui sont valides par rapport à une même architecture de conception *ArchiConception*. Celle-ci définit

une contrainte sur un paramètre du composant *B*. Lors de l'exécution, la valeur du paramètre peut être modifiée dans le cadre spécifié lors de la conception. Nous spécifierons dans le chapitre suivant un ensemble plus large de contraintes (paragraphe 5.3.5 page 136).

4.3.2 Variabilité pour l'architecture de déploiement

Comme nous l'avons expliqué précédemment, l'architecture de conception est celle qui contient le plus de variabilité. L'architecture de déploiement est un raffinement d'une architecture de conception ; elle contient donc moins de variabilité que l'architecture de conception. L'architecture de déploiement est utilisée afin de fournir à la plate-forme d'exécution l'état initial de l'application. Par conséquent, elle ne contient que la variabilité supportée par la plate-forme d'exécution. Dans le cadre de cette thèse, une architecture de déploiement a les caractéristiques suivantes :

- la topologie de l'application est fixée ;
- cette architecture ne présente plus de spécification, mais uniquement des implantations de composants ;
- la valeur des paramètres de configuration est déterminée.

Un niveau de variabilité est tout de même laissé au niveau de la sélection dynamique des services tiers requis par les composants [MCL+12, CLM+12].

4.3.3 Synthèse

Notre approche propose une prise en compte de la variabilité à quatre niveaux : topologique, substitution de composants, contrainte sur les composants et sélection de service.

Au niveau topologique, la variabilité est portée par les cardinalités des liaisons de l'architecture de conception. Celles-ci peuvent autoriser des instanciations multiples de liaisons et donc de composants.

La substitution d'une instance de composant par une autre au regard de leurs implantations respectives est permise par une référence commune des implantations vers une unique spécification de composant. Cela autorise la substitution de composants au sein d'une architecture sans en modifier la topologie.

Un troisième niveau de variabilité est porté par l'ajout de contraintes au niveau des composants. Nous avons donné un exemple en contraignant la valeur d'un paramètre. Nous verrons dans le chapitre suivant qu'il est possible de définir des contraintes sur des propriétés, des paramètres et des variables d'état.

Le dernier niveau de variabilité concerne les dépendances de services tiers. Nous verrons que ce cas s'applique particulièrement bien aux ressources externes qui peuvent être réifiées sous forme de services.

4.4 Conclusion

Ce premier chapitre de notre contribution nous a permis de mettre en place les lignes directrices de notre approche. Celle-ci repose sur un ensemble de modèles (et de méta-modèles) architecturaux en relation :

- **l'architecture de conception** est définie lors de la phase du même nom afin de rassembler toutes les décisions de conception et de déterminer ainsi l'ensemble des exécutions valides ;
- **l'architecture de déploiement** décrit la configuration qui devra être instanciée au démarrage du système ;
- **l'architecture de l'exécution** réifie les phénomènes d'exécution afin d'offrir un point de vue cohérent et de haut niveau sur l'exécution.

Entre ces différentes architectures, nous avons défini des relations en termes de transformation. De plus, nous avons souligné l'importance de pouvoir vérifier que l'architecture de l'exécution est valide par rapport à son architecture de conception. Pour cela, notre proposition détaillera au chapitre suivant les algorithmes nécessaires.

Chaque modèle dispose de caractéristiques propres en termes de moment de définition et d'utilisation dans le cycle de vie des applications. Celles-ci sont détaillées au tableau 4.1 page 105. Une caractéristique clé des modèles concerne la variabilité. C'est elle qui permet une évolution de l'architecture de l'exécution dans un cadre contraint. Pour cela, nous avons introduit la variabilité à quatre niveaux : topologique, substitution de composants, contrainte sur les composants et sélection de service.

Après cette présentation générale, le chapitre qui suit va s'attacher à formaliser de manière précise les modèles et méta-modèles architecturaux, ainsi que les algorithmes pour la validation de l'architecture de l'exécution.

Chapitre 5

Méta-modèles, modèles et algorithmes

Sommaire

5.1 Définitions des éléments architecturaux 112
5.1.1 Les types de composants 112
5.1.2 Les composants 114
5.1.3 Caractéristiques des composants et de leurs types 115
5.1.4 Les contraintes 118
5.1.5 Synthèse 120
5.2 Formalisation des architectures 122
5.2.1 Méta-modèle commun aux architectures 122
5.2.2 Méta-modèle de l'architecture de conception 124
5.2.3 Méta-modèle de l'architecture de déploiement 126
5.2.4 Méta-modèle de l'architecture de l'exécution 127
5.2.5 Synthèse 128
5.3 Formalisation des définitions et de leurs valeurs 129
5.3.1 Expression des définitions 129
5.3.2 Expression des propriétés 130
5.3.3 Expression des paramètres 131
5.3.4 Expression des variables d'état 134
5.3.5 Expression des contraintes 136
5.4 Validation des architectures 138
5.4.1 Motivations et démarche 138
5.4.2 Définition des objets mathématiques utilisés 139
5.4.3 Algorithme de correspondance 142
5.4.4 Algorithme de vérification 149
5.4.5 Synthèse 153
5.5 Conclusion 154

5.1 Définitions des éléments architecturaux

Dans les architectures présentées au chapitre précédent, nous avons utilisé de manière générale le terme de *composant*. A présent, nous allons établir une différence entre *type de composant* et *objet composant*. Par la suite, le terme composant réfèrera uniquement à l'objet composant.

Chaque architecture ne contient que des (objets) composants. Les types de composants sont élaborés en dehors des architectures de manière indépendante. Il existe une relation d'instanciation entre le composant et le type de composant. Dans le cadre de cette thèse, cette relation d'instanciation n'est pas uniquement présente à l'exécution comme c'est le cas habituellement dans certains langages. La figure 5.1 illustre les différentes notions : type de composant, composant et la relation d'instanciation.

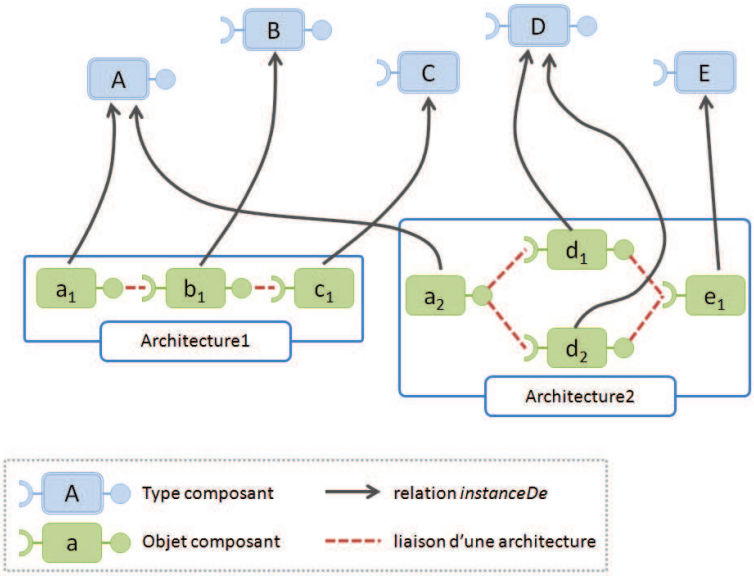

FIGURE 5.1 – Exemple introductif aux types de composant et aux composants.

L'intérêt de cette séparation entre type de composant et composant est double. Tout d'abord, il est possible d'instancier un même type dans plusieurs architectures sans avoir besoin de redéfinir celui-ci. C'est le cas du type A dans l'exemple de la figure 5.1. Le second intérêt est d'offrir la possibilité de disposer de plusieurs composants d'un même type au sein d'une même architecture (par exemple, d_1 et d_2).

Dans les deux sections suivantes, nous allons détailler les caractéristiques des types de composants puis des composants.

5.1.1 Les types de composants

Nous définissons deux types de composants : les implantations et les spécifications de composants. Une implantation est le code exécutable (éventuellement compilé) d'un composant. Notons que ce code peut ne pas être suffisant pour instancier le composant qui peut

avoir besoin pour cela d'une configuration, comme nous le verrons par la suite. Outre le séquençage des traitements, l'implantation définit aussi les capacités d'interactions du composant avec les autres composants par l'intermédiaire d'interfaces que nous décrirons par la suite.

Les spécifications de composants, quant à elles, ne se focalisent que sur la description des capacités de communication des composants, sans préciser la manière avec laquelle les traitements sont effectués. Les préoccupations abordées par les spécifications de composants sont donc un sous-ensemble des préoccupations traitées par les implantations. Pour cette raison, il est possible de définir une relation implante entre une implantation de composant et une spécification de composant. Nous dirons qu'une implantation implante une spécification si et seulement si l'implantation et la spécification de composant disposent exactement des mêmes capacités de communication avec les autres composants.

Pour permettre aux composants de communiquer, ceux-ci disposent d'interfaces. Ces dernières peuvent être soit fournies, soit requises. Dans le premier cas, le composant offre une fonctionnalité décrite par une description de service. Dans le second, le composant a besoin d'un service pour pouvoir fonctionner, lui aussi décrit par une description de service. Une architecture est construite en établissant des liaisons entre des interfaces fournies et requises de composants différents. Pour que cette liaison puisse être établie, il est nécessaire que les interfaces liées présentent des contrats de services compatibles ; c'est-à-dire que le contrat de l'interface requise doit être inclus dans le contrat de l'interface fournie. Dans notre cas, nous exigerons simplement l'identité des contrats.

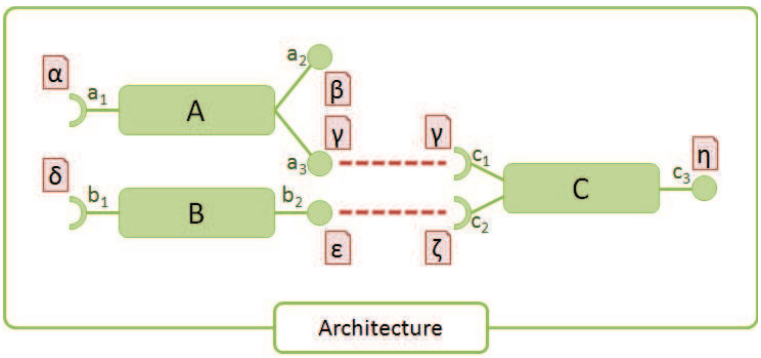

FIGURE 5.2 – Illustration des interfaces et des contrats entre composants.

L'architecture de la figure 5.2 dispose de trois composants : A, B et C. Le composant A dispose d'une interface requise (a_1) qui nécessite un service de description α offert par un autre composant. A dispose aussi de deux interfaces fournies, a_2 et a_3, qui fournissent respectivement des services de description β et γ. De manière analogue, le composant B dispose d'une interface requise (b_1, avec la description δ) et d'une interface fournie (b_2, avec la description ε) et le composant C de deux interfaces requises (c_1 avec la description γ et c_2, avec la description ζ) et d'une interface fournie (c_3, avec la description η).

Deux liaisons ont été établies dans le cadre de cette architecture. La première lie l'interface fournie a_3 du composant A et l'interface requise c_1 du composant C. Cette liaison

est valide, car elle lie deux interfaces qui appartiennent à des composants différents et présentent des descriptions de services identiques. La seconde liaison est établie entre l'interface fournie b_2 du composant B et l'interface requise c_2 du composant C. Ici aussi, les deux interfaces appartiennent à des composants différents. Cependant, les descriptions de services étant différentes, il est nécessaire que la description ζ du service requis soit incluse dans la description ε du service fourni (mais nous ne traiterons pas par la suite ce cas particulier).

5.1.2 Les composants

Les composants peuvent être soit des configurations de spécification, soit des configurations d'implantation, soit des instances de composants. Comme le montre la figure 5.3, chaque composant instancie un type présenté précédemment.

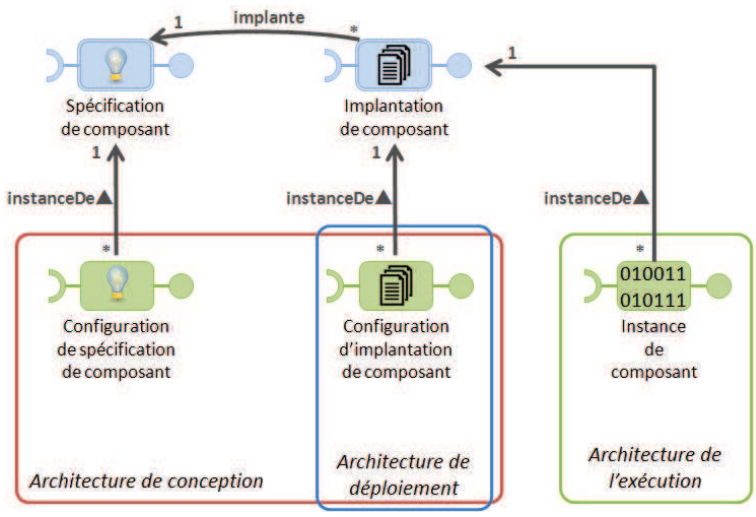

FIGURE 5.3 – Les composants utilisés dans les différentes architectures.

De plus, chaque composant peut être utilisé dans une architecture. Ainsi, dans les architectures de conception, il est possible d'assembler des configurations de spécifications et d'implantations de composants qui viennent respectivement configurer un type de spécification ou d'implantation. L'architecture de déploiement ne contient que des configurations d'implantation de composant puisque la variabilité exprimée dans l'architecture de conception a été supprimée pour ne laisser que celle supportée par la plate-forme d'exécution. L'architecture de l'exécution ne dispose que d'instances de composants. Pour nous, une instance de composant n'est pas à proprement parler du code exécuté. C'est plus précisément la réification de l'exécution d'une implantation de composant qui a été initialisée avec une configuration d'implantation de composant. Il s'agit donc d'une abstraction qui permet de rendre compte de l'exécution du système en offrant un point de vue selon l'approche à composants (dans notre cas, orientés services dynamiques). Cette instance conserve une relation *instanceDe* vers l'implantation du composant dont elle réifie une instance dans la plate-forme d'exécution.

5.1.3 Caractéristiques des composants et de leurs types

Nous allons, dans cette section, exposer les quatre concepts qui sont utilisés de manière différente selon les composants (composants et types composants) : les propriétés, les paramètres, les variables d'état et les contraintes.

5.1.3.1 Les propriétés

Nous avons vu précédemment que le lien entre spécification et implantation permettait de substituer à l'exécution une instance par une autre à condition que leurs implantations implantent une spécification commune de composant présente dans l'architecture de référence (paragraphe 4.3.1.2 page 107). Cependant, pour que ce mécanisme soit pertinent, il est nécessaire de pouvoir discriminer les implantations pour comprendre les spécificités de chacune. C'est là l'objectif des propriétés.

Les propriétés permettent de caractériser les implantations à l'aide de couples clé-valeur. Par exemple, si une implantation est sécurisée, elle pourra bénéficier d'une propriété de la forme *estSecurisé* = *vrai*. De plus, nous avons ajouté la possibilité de définir des propriétés au niveau des spécifications. De cette manière, toute propriété définie au niveau spécification de composant devra impérativement être valuée dans chacune de ses implantations.

L'exemple de la figure 5.4 nous présente ainsi une spécification et deux implantations. La spécification *spec1* définit deux propriétés (*estSécurisé* et *tpsMaxGaranti*) qui sont donc valuées dans ses deux implantations *impl1* et *impl2*. La seconde implantation dispose d'une propriété supplémentaire (*mémoireMaxGarantie*). Avec ces informations, il devient possible de choisir à l'exécution l'implantation la plus adaptée à un contexte d'exécution, qui dans notre cas pourra donner la priorité à la sécurité ou à la rapidité du traitement, selon le besoin.

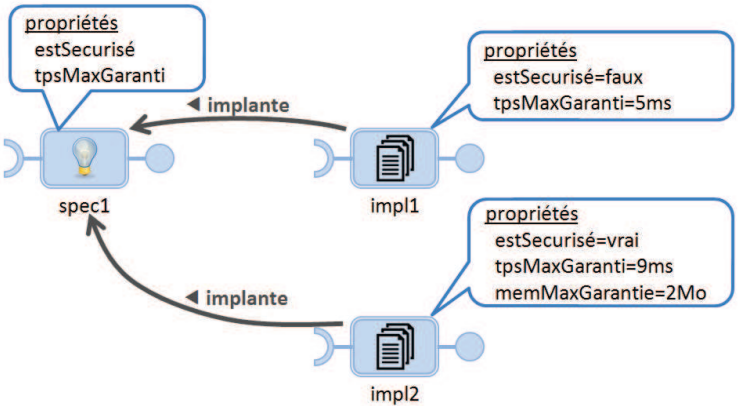

FIGURE 5.4 – Utilisation des propriétés pour les types de composants.

Considérons à présent les composants. Par le truchement de la relation d'instanciation, ceux-ci peuvent retrouver de manière indirecte les propriétés auprès de leur type. De plus, dans la mesure où les propriétés ne servent qu'à caractériser les implantations de composants, elles n'ont pas à être présentes (et encore moins, à être définies) au niveau des objets composants qui restent donc vierges de toutes propriétés (figure 5.5).

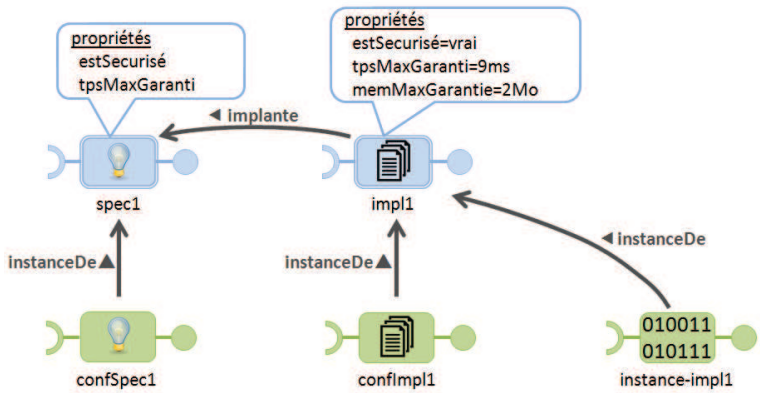

FIGURE 5.5 – Utilisation des propriétés avec les composants et leurs types.

5.1.3.2 Les paramètres

Les paramètres permettent d'ajuster la configuration ou le comportement d'une instance exécutée en jouant sur la valeur de variables. Pour comprendre la logique de leur utilisation, nous allons parcourir les architectures en commençant par l'exécution et en « remontant » vers la conception.

L'architecture de l'exécution contient un ensemble d'instances. Ces instances disposent de paramètres valués, puisque ceux-ci ont été prévus pour modifier leur configuration ou leur comportement. En conséquence, les paramètres sont nécessairement définis dans les implantations.

L'architecture de déploiement, constituée exclusivement à l'aide d'instances configurées de composants. Pour fournir les valeurs d'initialisation à l'exécution, il est absolument nécessaire que chaque paramètre dispose d'une valeur. Les configurations d'instances doivent donc disposer de paramètres valués.

Dans l'architecture de conception, une première configuration des paramètres peut être effectuée (et éventuellement modifiée dans l'architecture de déploiement afin d'ajuster les valeurs des paramètres à une cible particulière). Nous avons donc besoin des valeurs au niveau des configurations de spécifications et aussi des configurations d'implantations. De plus, pour guider la configuration des spécifications, les paramètres devront être définis dans les types spécifications. Tout paramètre défini dans le type spécification doit être implanté dans les implantations qui implantent cette spécification.

Enfin, dans l'hypothèse où une instance est créée lors de l'exécution (et non seulement au démarrage de l'application), il est nécessaire de disposer de valeurs d'initialisation pour tous ses paramètres. Pour cela, tous les paramètres des types implantation de composant recevront des valeurs qui pourront le cas échéant servir de valeur par défaut lors de l'instanciation à l'exécution.

Un exemple résumant ce que nous venons de voir se trouve à la figure 5.6. Sur celle-ci, nous pouvons voir que la spécification définit deux paramètres : *serverURL* et *nbThreads*. Ces

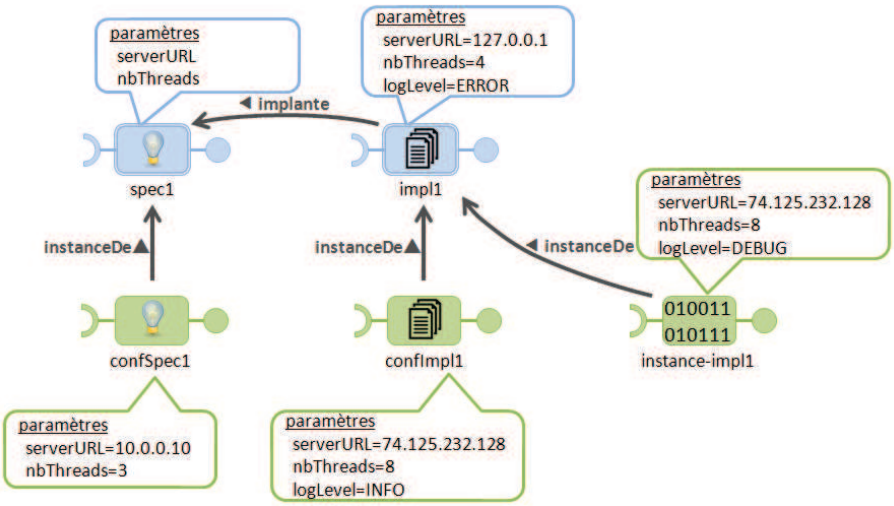

FIGURE 5.6 – Utilisation des paramètres.

deux paramètres sont donc implantés avec des valeurs par défaut au niveau de l'implantation *impl1*. Par ailleurs, cette dernière ajoute un troisième paramètre, *logLevel*, qui reçoit lui aussi une valeur par défaut.

Dans une architecture de conception, nous pourrions avoir aussi bien des configurations de spécifications que d'implantations, toutes deux avec des valeurs par défaut de paramètres. Ces valeurs pourraient être utilisées lors de la création de l'architecture de déploiement, qui pour sa part ne serait constituée qu'à partir de configurations d'implantations. Dans cette architecture, tous les composants disposeraient de valeurs d'initialisation pour tous les paramètres. Enfin, l'architecture de l'exécution aurait, elle aussi, des valeurs pouvant évoluer au fil du temps pour tous les paramètres de tous les composants exécutés, à l'image de *instance-impl1*.

5.1.3.3 Les variables d'état

Les variables d'état ont pour objectif de rendre compte du déroulement de l'exécution. Leurs valeurs ne sont donc accessibles que dans les instances de composants et donc uniquement lors de l'exécution. Les variables d'état peuvent être définies dans les types spécification et implantation de composants. Notons que celles-ci sont totalement absentes des configurations de spécification et d'implantation.

Sur l'exemple de la figure 5.7, la spécification de composant *spec1* définit une variable d'état *tpsTraitementMoyen* qui permet de rendre compte du temps moyen de traitement de chaque message. L'implantation ajoute une seconde variable d'état, qui permet de compter le nombre de messages traités lors de l'exécution (*cptMessage*). Enfin, ces deux variables disposent de valeurs mises à jour lors de l'exécution de *instance-impl1* et alors accessibles uniquement en lecture pour rendre compte des phénomènes d'exécution.

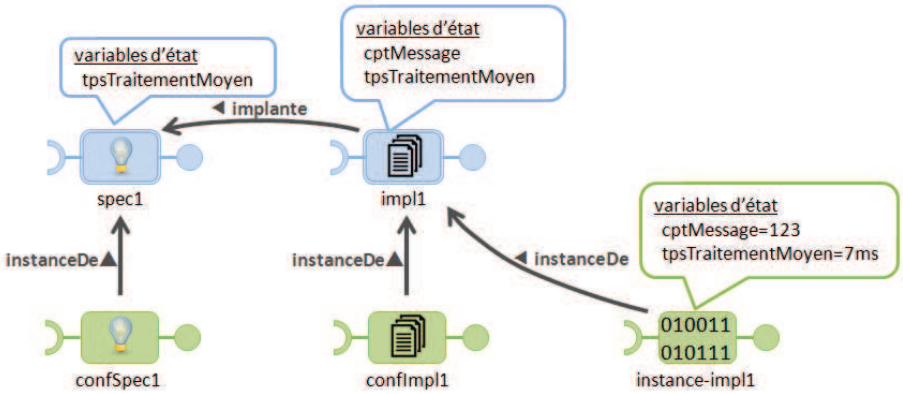

FIGURE 5.7 – Utilisation des variables d'état.

5.1.4 Les contraintes

Les contraintes ne sont présentes que dans l'architecture de conception. Leur objectif est de venir restreindre la variabilité en ajoutant des règles qui doivent être satisfaites à chaque instant. Ces contraintes peuvent être appliquées aux valeurs des propriétés, des paramètres ou des variables d'état.

5.1.4.1 Les contraintes sur les propriétés

Les contraintes appliquées aux propriétés permettent de restreindre le choix des implantations correspondant à une spécification de composant présente dans une architecture de conception.

Dans l'exemple de la figure 5.8, nous avons une architecture de conception qui dispose d'une configuration de spécification appelée *confSpec1*. Cette configuration instancie la spécification *spec1*. Dans l'architecture de l'exécution, il est donc théoriquement possible de disposer d'instances de n'importe quelle implantation qui implante *spec1*. La contrainte définie sur la configuration de spécification vient limiter ce choix. Elle impose que l'implantation dispose d'une propriété *estSécurisé=vrai*, ce qui est le cas dans notre exemple avec l'implantation *impl1*. Notons qu'il n'est pas possible de définir de contrainte sur les propriétés des configurations d'implantations de l'architecture de conception car cela n'aurait aucun sens. Notons aussi qu'une vérification des contraintes sur les propriétés peut éventuellement être effectuée au niveau de l'architecture de déploiement de manière analogue.

5.1.4.2 Les contraintes sur les paramètres

Contrairement aux contraintes sur les propriétés, les contraintes sur les paramètres peuvent être aussi bien définies au niveau des configurations de spécification que des configurations d'implantations de l'architecture de conception. Sur l'exemple de la figure 5.9, une contrainte sur la spécification *confSpec1* a été définie afin de restreindre les valeurs valides du paramètre *nbThreads* en obligeant celles-ci à être inférieures ou égales à 4.

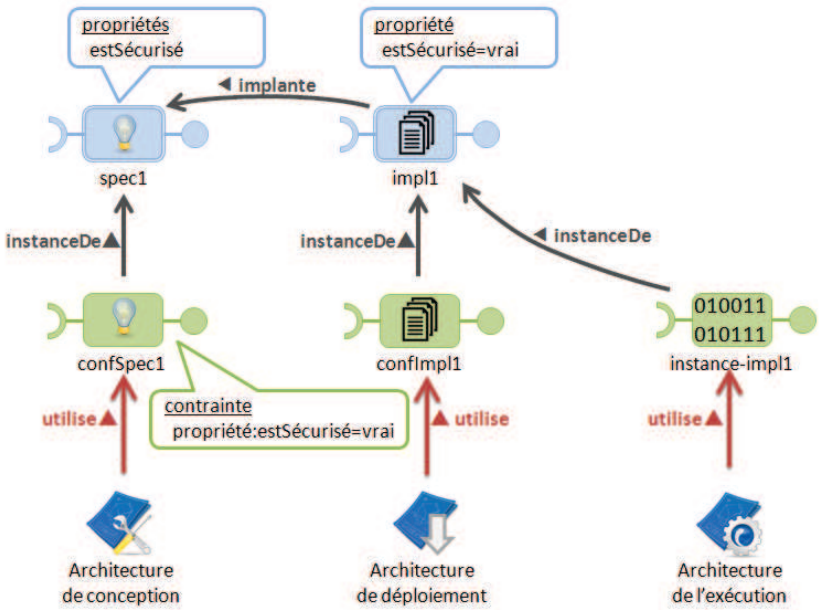

FIGURE 5.8 – Exemple d'utilisation de contrainte sur une propriété.

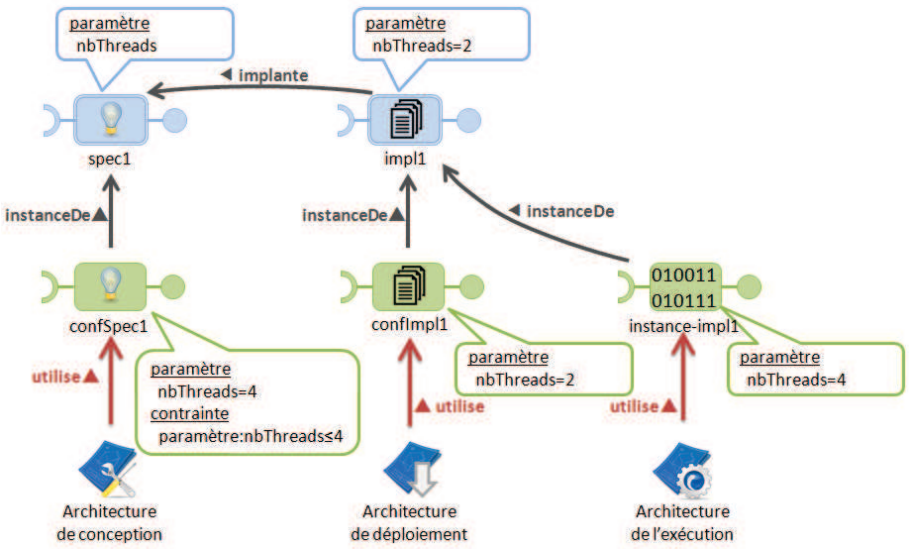

FIGURE 5.9 – Exemple d'utilisation de contrainte sur un paramètre.

5.1.4.3 Les contraintes sur les variables d'état

Pour terminer, nous allons présenter le fonctionnement des contraintes appliquées aux variables d'état. Celles-ci peuvent être définies aussi bien au niveau des configurations de spécification que des configurations d'implantation de l'architecture de conception. Elles conditionnent ainsi la validité de l'architecture de l'exécution à la valeur des variables d'état, qui sont mises à jour en continu lors de l'exécution.

Dans l'exemple de la figure 5.10, une architecture de conception dispose d'une configuration de composant *confSpec1*, qui instancie la spécification *spec1*. Cette dernière définissant une variable d'état *tpsTraitementMoyen*, il est possible d'appliquer à *confSpec1* une contrainte sur les valeurs de celle-ci. En conséquence, l'architecture deviendra invalide à l'exécution si la valeur de cette variable d'état dépasse *20ms*. Notons que ces contraintes peuvent aussi être définies de la même manière sur les configurations d'implantations de l'architecture de conception.

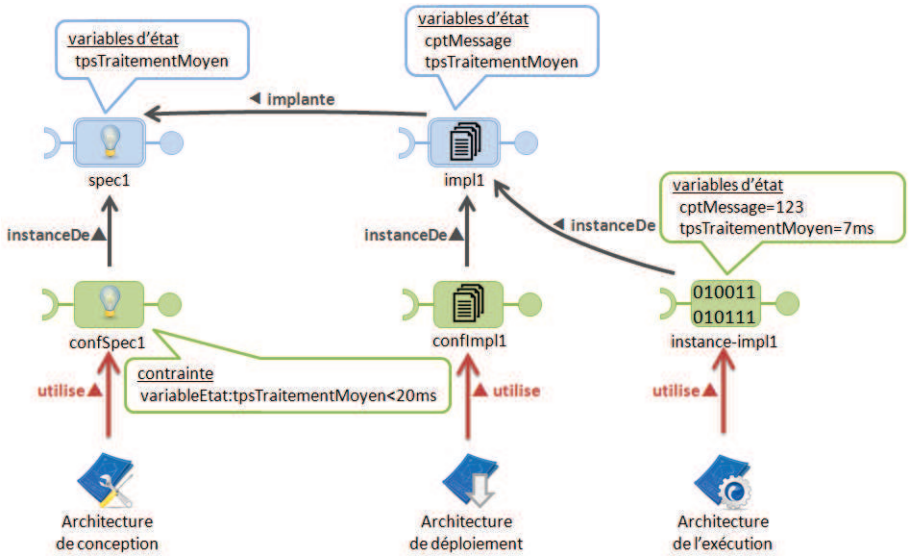

FIGURE 5.10 – Exemple d'utilisation de contrainte sur une variable d'état.

5.1.5 Synthèse

Suite à la présentation générale du chapitre précédent, nous venons d'approfondir les concepts de composant et d'architecture. Cela nous a permis de faire une distinction entre les *types* de composants et les (*objets*) composants. De plus, nous avons détaillé les concepts de propriété, de paramètre et de variable d'état. Les propriétés permettent de caractériser les implantations. De cette manière, il devient possible de substituer un composant par un autre au regard des caractéristiques propres de leur implantations. Les paramètres permettent de modifier le comportement des instances lors de l'exécution. Enfin, les variables d'état permettent de rendre compte du déroulement de l'exécution à l'aide de variables mises à jour en permanence. En prenant appui sur ces propriétés, paramètres et variables d'état, nous

avons introduit un système de contraintes permettant de restreindre la variabilité introduite précédemment.

Le tableau 5.1 met en avant la séparation des composants en types de composants et en composants. Les types de composants (spécifications et implantations de composants) ne peuvent pas être utilisés directement dans les architectures, contrairement aux composants dont c'est justement la fonction. Ainsi, les configurations de spécification de composant ne peuvent être utilisées que dans les architectures de conception, tout comme les configurations d'implantation de composant qui peuvent de plus trouver leur place dans les architectures de déploiement. Enfin, les instances de composants ne peuvent être employées que dans les architectures de l'exécution.

	TYPES DE COMPOSANTS		COMPOSANTS		
	Spécification de composant	**Implantation de composant**	**Configuration de spécification de composant**	**Configuration d'implantation de composant**	**Instance de composant**
Utilisation directe dans…	/	/	architecture de conception	architecture de conception et de déploiement	architecture de l'exécution
Propriétés	définitions	définitions et valeurs	/	/	/
Paramètres	définitions	définitions et valeurs	définitions et valeurs	définitions et valeurs	définitions et valeurs
Variables d'état	définitions	définitions	/	/	définitions et valeurs en lecture

Tableau 5.1 – Comparaison des composants et de leurs types.

Concernant les propriétés, les paramètres et les variables d'état, nous avons distingué les définitions des valeurs. Par définition, nous entendons la déclaration seule, là où une valeur value cette déclaration.

5.2 Formalisation des architectures

Dans la première partie de ce chapitre, nous avons étudié les composants et leurs types. Nous allons à présent nous intéresser à leur utilisation par les architectures. Nous rappelons que ces dernières ne sont formées qu'à partir de composants, alors que les types existent en dehors des architectures (figure 5.1 page 112). La présentation des architectures va passer par l'explication de leurs méta-modèles respectifs. Pour cela, nous allons commencer par exposer ce qui est commun aux trois méta-modèles (paragraphe 5.2.1). Ensuite, nous traiterons successivement de l'architecture de conception (paragraphe 5.2.2 page 124), de l'architecture de déploiement (paragraphe 5.2.3 page 126) et de l'architecture de l'exécution (paragraphe 5.2.4 page 127).

5.2.1 Méta-modèle commun aux architectures

Le méta-modèle commun aux trois architectures est présenté à la figure 5.11.

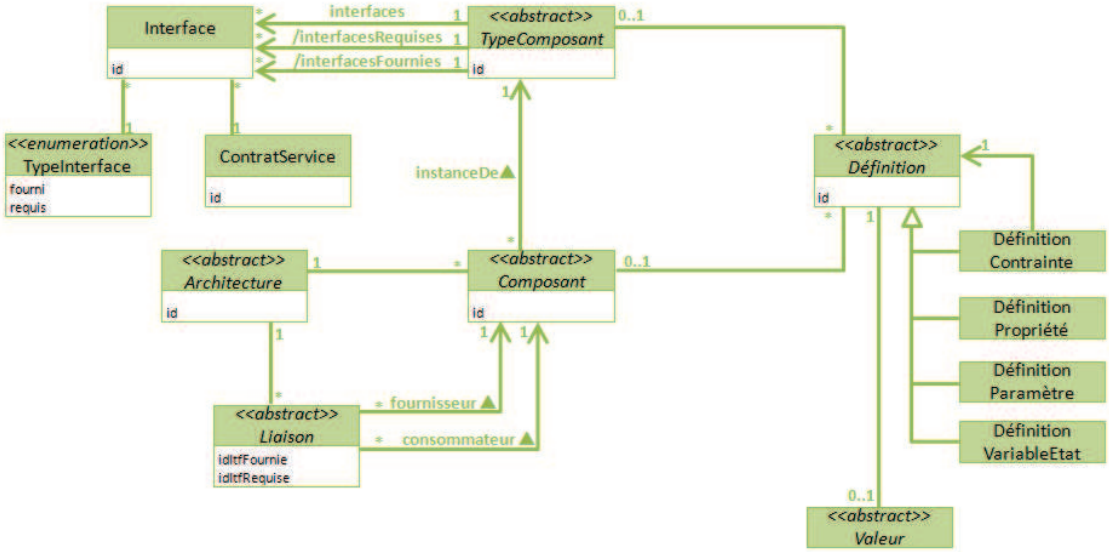

FIGURE 5.11 – Méta-modèle commun définissant une architecture.

Une architecture est représentée par la classe abstraite *Architecture*. Chaque architecture est définie par un identifiant (*id*) unique, des composants et des liaisons. Chaque liaison et chaque composant a nécessairement une unique architecture d'appartenance. Chaque composant est défini par un identifiant (*id*) tel que tous les composants, qui font partie d'une même architecture, ont tous des identifiants différents. Chaque composant est instance d'un type de composant, défini par un identifiant (*id*) unique. Chaque type de composant dispose d'un ensemble d'interfaces, qui ne peuvent pas être partagées entre composants. Chaque interface est identifiée avec un *id*, de sorte qu'un type de composant ne puisse pas disposer de deux interfaces de même identifiant.

Une interface peut être soit fournie, soit requise, comme l'indique son *TypeInterface*. Une interface fournie permet à un composant d'offrir un service, alors qu'une interface requise créé au contraire une dépendance de service. Les différentes interfaces d'un type de composant peuvent donc être réparties entre les interfaces fournies et les interfaces requises. C'est la raison pour laquelle nous avons les relations dérivées *interfacesRequises* et *interfacesFournies* (contrainte OCL 5.1).

```
context TypeComposant
    inv: self.interfaces → select(i | i.TypeInterface = requis) →
        forAll(i | self.interfacesRequises → exists(ir | ir=i))
    inv: self. interfacesRequises →
        forAll(i | self.interfaces → select(i | i.TypeInterface = requis) → exists(ir | ir=i))
    inv: self.interfaces → select(i | i.TypeInterface = fourni) →
        forAll(i | self.interfacesFournies → exists(if | if=i))
    inv: self.interfacesFournies →
        forAll(i | self.interfaces → select(i | i.TypeInterface = fourni) → exists(if | if=i))
```

Contrainte OCL 5.1: Relations entre *interfacesRequises* et *interfacesFournies*.

Quel que soit le type de l'interface, le service dont il est question, est décrit grâce au *ContratService* de l'interface. Chaque *ContratService* dispose lui aussi d'un identifiant unique.

```
context Liaison
    inv: self.fournisseur.instanceDe.interfacesFournies → one(i | i.id=self.idItfFournie)
    inv: self.consommateur.instanceDe.interfacesRequises → one(i | i.id=self.idItfRequise)
```

Contrainte OCL 5.2: Types des interfaces pour les liaisons.

Une liaison permet de lier deux composants par l'intermédiaire de leurs interfaces, définies au niveau de leur type. C'est la raison pour laquelle une liaison est définie par les deux composants ainsi que par les identifiants des interfaces (présents dans leur type) à lier (contrainte OCL 5.2).

```
context Liaison
    inv: self.fournisseur.instanceDe.interfacesFournies →
        select(i | i.id = self.idItfFournie) → asSequence() →
            first().ContratService.id = self.consommateur.instanceDe.interfacesRequises →
                select(i | i.id = self.idItfRequise) → asSequence() → first().ContratService.id
    inv: self.fournisseur <> self.consommateur
context Architecture
    inv: self.Liaison → forAll(l1, l2 | l1 <>l2
    implies (l1.fournisseur <> l2.fournisseur)
        or (l1.consommateur <> l2.consommateur)
        or (l1.idItfFournie <> l2.idItfFournie)
        or (l1.idItfRequise <> l2.idItfRequise))
```

Contrainte OCL 5.3: Contraintes sur les liaisons.

A cela, il faut encore ajouter trois contraintes supplémentaires. Tout d'abord, une liaison ne peut lier que des interfaces qui exposent le même contrat de service. De plus, il est impossible d'établir une liaison entre deux interfaces d'un même composant. Enfin, il n'est pas possible de trouver deux liaisons reliant exactement les mêmes interfaces au sein d'une même architecture (contrainte OCL 5.3).

Nous avons vu que les composants pouvaient disposer de propriétés, de paramètres de variables d'état et de contraintes, éventuellement valués selon le type de composant considéré. Pour cette raison, nous avons séparé l'aspect définition de la valeur pour ces trois concepts. Une définition est en relation soit avec un *TypeComposant*, soit avec un *Composant* (de manière exclusive). De plus, toutes les définitions d'un composant ou d'un type de composant ont des identifiants uniques au sein de celui-ci. Une définition de contrainte est en relation avec une définition, qui ne peut pas être une contrainte. Nous avons donc la contrainte OCL 5.4.

context DéfinitionContrainte
  inv: not(self.Définition.oclIsTypeOf(DéfinitionContrainte))

Contrainte OCL 5.4: Non récursivité des définitions de contraintes.

Enfin, une définition peut être ou non liée à une valeur. Nous approfondirons ce point par la suite.

5.2.2 Méta-modèle de l'architecture de conception

L'architecture de conception est formalisée avec le méta-modèle présenté dans la figure 5.12. Ce méta-modèle étend le méta-modèle commun (figure 5.11 page 122).

Ce méta-modèle représente les concepts nécessaires pour définir des modèles de l'architecture de conception. Nous devons retrouver dans ce méta-modèle les éléments qui sont particuliers à l'architecture de conception : les composants sont soit des spécifications, soit des implantations. Cette architecture peut disposer de configurations de spécifications et de configurations d'implantations de composant. Comme nous l'avons vu, une configuration de spécification (respectivement d'implantation) de composant instancie forcément une spécification (respectivement une implantation) de composant. Cela est exprimé avec la contrainte OCL 5.5.

context ArchitectureConception
  inv: self.Composant → forAll(c.oclIsTypeOf(ConfSpécificationComposant)
    or c.oclIsTypeOf(ConfImplantationComposant))
context ConfSpécificationComposant
  inv: self.référence.oclIsTypeOf(SpécificationComposant)
context ConfImplantationComposant
  inv: self.référence.oclIsTypeOf(ImplantationComposant)

Contrainte OCL 5.5: Contraintes sur les types de composants des architectures de conception.

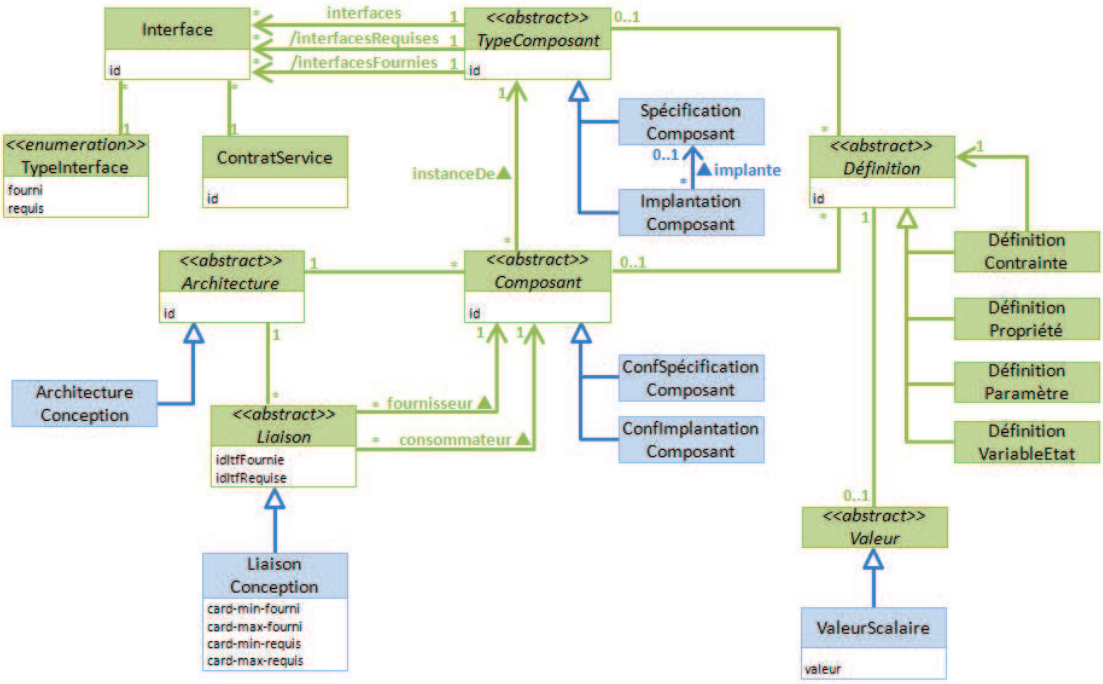

FIGURE 5.12 – Méta-modèle de l'architecture de conception.

De plus, cette architecture supporte de la variabilité exprimée au niveau des liaisons avec des cardinalités. Ces dernières sont présentes sur chacune des extrémités des liaisons : côté interface fournie (*card-min-fourni* et *card-max-fourni*) et côté interface requise (*card-min-requis* et *card-max-requis*).

Considérons à présent les composants. Une architecture de conception peut disposer de configurations de spécifications et de configurations d'implantations de composant. Comme nous l'avons vu, une configuration de spécification (respectivement d'implantation) de composant instancie forcément une spécification (respectivement d'implantation) de composant. De plus, une liaison permet d'exprimer qu'une implantation de composant peut ou non implanter une unique spécification de composant. Cette liaison est orientée, car la spécification n'a pas besoin de savoir quelles sont les implantations qui l'implantent.

Notons que dans l'architecture de conception, la classe abstraite *Valeur* est étendue par la classe *ValeurScalaire*. Nous approfondirons cela dans un développement séparé, à la suite de la présentation des différentes architectures. Nous pouvons donc dès à présent formaliser l'architecture de déploiement.

5.2.3 Méta-modèle de l'architecture de déploiement

Tout comme le méta-modèle de l'architecture de conception, le méta-modèle de l'architecture de déploiement (figure 5.13) étend le méta-modèle commun (figure 5.11 page 122). De plus, une architecture de déploiement peut référencer une architecture de conception. L'indirection indiquée sur le méta-modèle permet de séparer le méta-modèle de déploiement de celui de conception. Celle-ci n'est navigable que dans un sens, car une architecture de conception n'a pas à savoir par quelles autres architectures elle est utilisée.

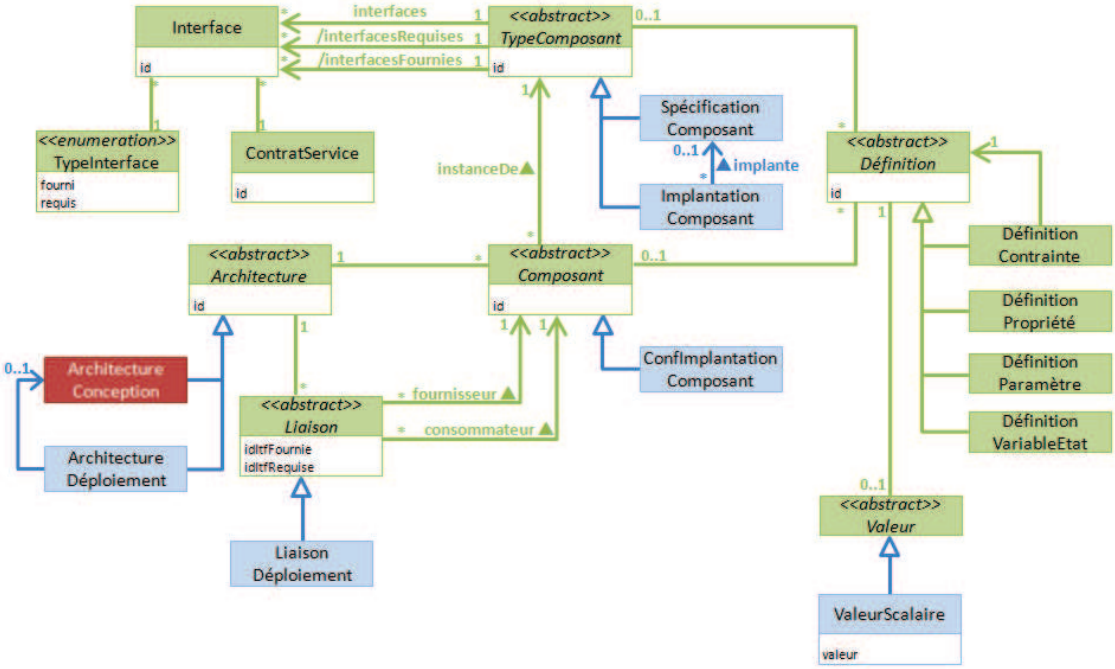

FIGURE 5.13 – Méta-modèle de l'architecture de déploiement.

Côté composants, une architecture de conception ne peut disposer que de configurations d'implantations de composants, chacune instanciant une implantation de composant. Cette restriction est exprimée avec la contrainte OCL 5.6.

context ArchitectureDéploiement
  inv: self.Composant → forAll(c.oclIsTypeOf(ConfImplantationComposant))
context ConfImplantationComposant
  inv: self.instanceDe.oclIsTypeOf(ImplantationComposant)

Contrainte OCL 5.6: Contraintes sur les types de composants d'une architecture de déploiement.

Enfin, tout comme dans l'architecture de conception, la classe abstraite *Valeur* est étendue par la classe *ValeurScalaire*.

5.2.4 Méta-modèle de l'architecture de l'exécution

Le méta-modèle de l'architecture de l'exécution (figure 5.14) vient, elle-aussi, étendre le méta-modèle commun (figure 5.11 page 122). Tout comme les architectures de déploiements, les architectures de l'exécution peuvent référencer une architecture de conception via une indirection. Ici aussi, cette dernière n'est navigable que dans un sens, car une architecture de conception n'a pas à savoir par quelles autres architectures elle est utilisée.

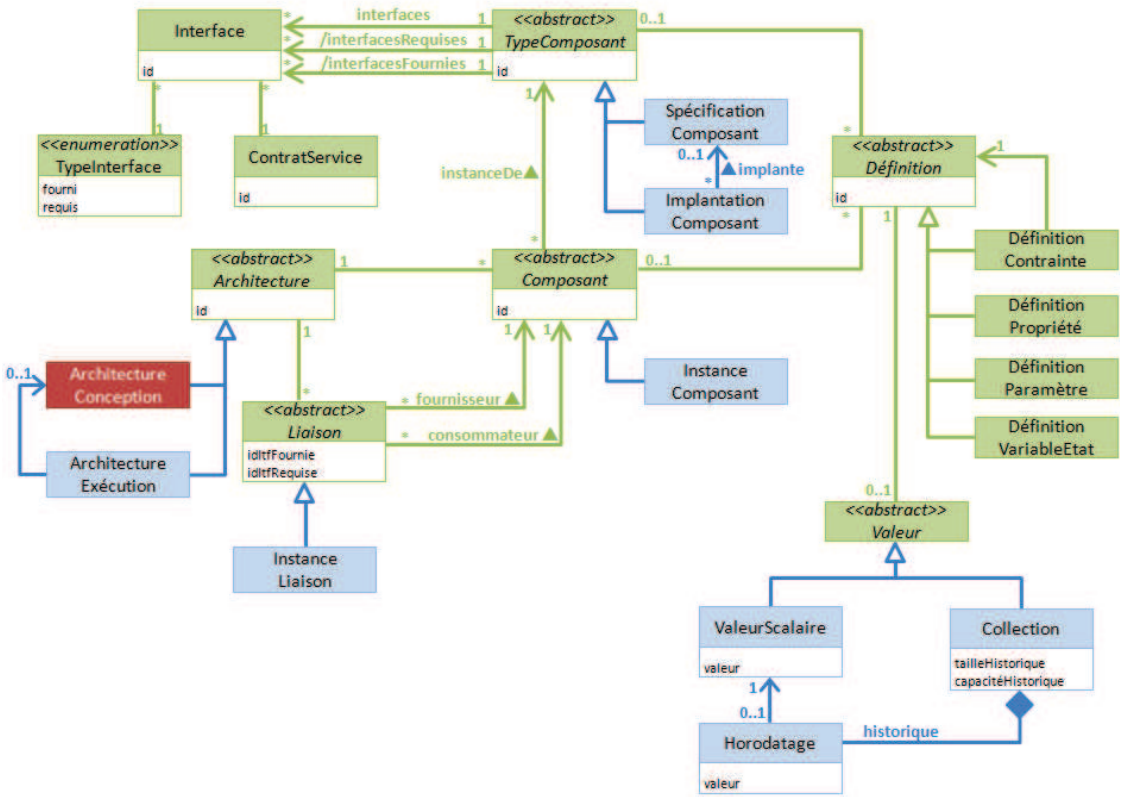

FIGURE 5.14 – Méta-modèle de l'architecture de l'exécution.

Comme nous l'avons vu, une architecture de l'exécution ne peut disposer que d'instances de composants (exprimée par la contrainte OCL 5.7), chacune instanciant une implantation de composant.

context ArchitectureExécution
  inv: self.Composant → forAll(c.oclIsTypeOf(InstanceComposant))
context InstanceComposant
  inv: self.instanceDe.oclIsTypeOf(ImplantationComposant)

Contrainte OCL 5.7: Contraintes sur les types de composants des architectures de l'exécution.

Notons enfin que la hiérarchie de classes issue de la classe *Valeur* est différente par rapport aux deux autres architectures. En effet, dans le cas des variables d'état, nous souhaitons

disposer d'un historique horodaté des valeurs successives prises par celle-ci. La classe *Collection* permet donc d'agréger un ensemble de valeurs scalaires, par l'intermédiaire de la classe horodatage. Nous allons à présent détailler ce point en formalisant les relations entretenues entre les composants, les définitions et les valeurs.

5.2.5 Synthèse

Nous venons de décrire les trois méta-modèles d'architectures de notre proposition. Nous avons vu que ceux-ci présentent des variations autour d'un méta-modèle commun (figure 5.11 page 122). Dans celui-ci une architecture dispose d'un ensemble de composants en relation par des liaisons. De plus, chaque composant est en relation avec un *TypeComposant* par une relation *instanceDe*. Les classes *TypeComposant* et *Composant* peuvent disposer de définitions, évenuellement valuées.

Sur la base du méta-modèle commun, les méta-modèles des différentes architectures sont conçus en ajoutant des sous-classes aux classes *Architecture*, *Liaison*, *Composant*, *TypeComposant* et *Valeur* du méta-modèle commun. Nous retouvons ainsi la relation d'héritage présentée à la figure 4.3 page 102.

Après cette présentation générale des méta-modèles, nous allons maintenant nous attacher à décrire un point clé de notre contribution : la partie de ceux-ci relative aux classes *Définition* et *Valeur*. Celles-ci servent de support à la définition et à la valuation des propriétés, des paramètres, des variables d'états et des contraintes.

5.3 Formalisation des définitions et de leurs valeurs

5.3.1 Expression des définitions

Le figure 5.15 présente l'agrégation de l'ensemble des variations du méta-modèle commun qui servent à exprimer les définitions et leurs valeurs. Il permettra de mieux comprendre les contraintes que nous allons exprimer.

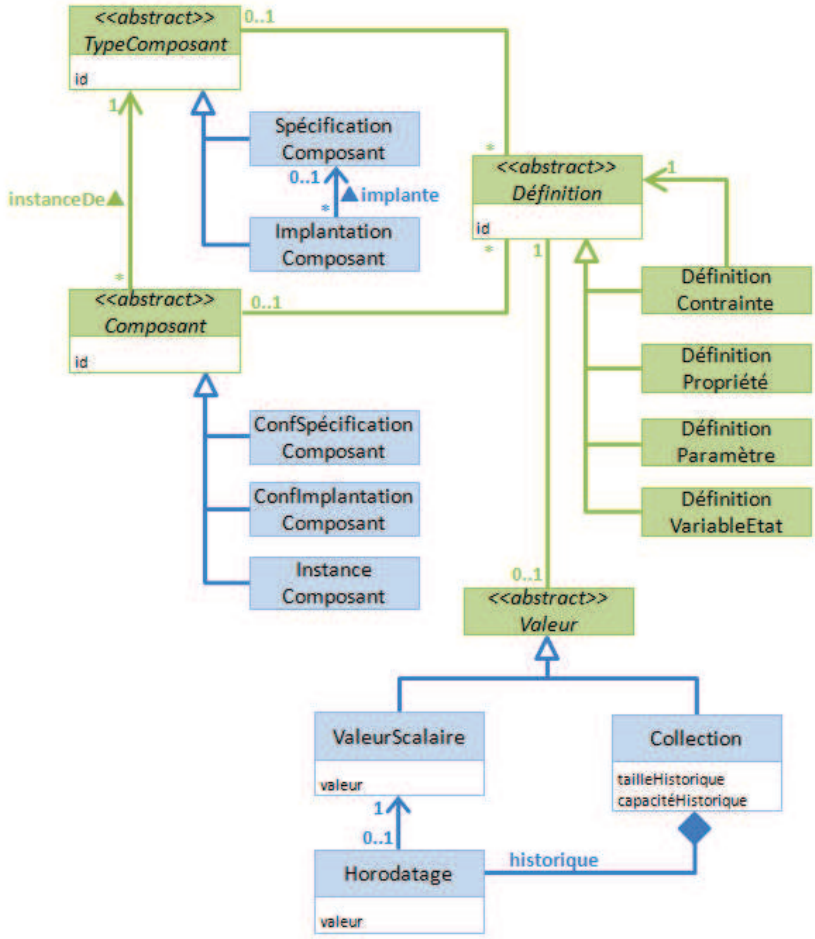

FIGURE 5.15 – Ensemble des variations possibles autour des classes *Définition* et *Valeur* du méta-modèle commun.

Dans le méta-modèle (figure 5.15), nous disposons d'une classe abstraite *Définition*, qui sert de classe de base et qui est héritée par *DéfinitionContrainte*, *DéfinitionPropriété*, *DéfinitionParamètre* et *DéfinitionVariableEtat*. Une définition est caractérisée par un identifiant. Chaque composant peut disposer d'un nombre quelconque de définitions et chaque définition appartient à un unique composant. Les définitions peuvent disposer, selon le cas, d'une valeur. Nous avons donc un lien depuis la classe *Définition* vers une classe *Valeur*, tout en prévoyant que cette dernière puisse être optionnelle. Chaque valeur n'est en relation qu'avec une unique définition.

Notons que ces valeurs peuvent être soit scalaires, soit des collections. Dans le premier cas, nous disposons d'une unique valeur. Dans le second, d'un ensemble de valeurs successives et horodatées qui permettent de constituer un historique. Celui-ci dispose d'une taille fixe et les valeurs les plus anciennes sont donc écrasées au profit de valeurs plus récentes quand cet historique est plein. Nous avons donc la classe *Valeur*, qui est héritée par les classes *ValeurScalaire* et *Collection*. Cette dernière dispose d'un ensemble d'horodatages, chacun d'eux ayant sa propre valeur scalaire.

5.3.2 Expression des propriétés

Toute spécification de composant peut disposer d'un ensemble de définitions de propriétés. Celles-ci ne disposent pas de valeurs (contrainte OCL 5.8) puisque ces dernières permettent de caractériser les implantations. Dans notre exemple (figure 5.5 page 116), *spec1* dispose de deux définitions de propriété nommées *estSécurisé* et *tpsMaxGaranti*.

```
context SpécificationComposant
    inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionPropriété)) →
        forAll(dp : DéfinitionPropriété | dp.Valeur → isEmpty())
```

Contrainte OCL 5.8: Définition de propriété au niveau spécification de composant.

Chacune des implantations doit disposer des définitions de propriétés définies au niveau de sa spécification, si elle en a une. De plus, les implantations peuvent définir leurs propres définitions de propriétés. C'est ici le cas avec la propriété qui a pour identifiant *memMaxGarantie*. Dans les deux cas, ces définitions doivent disposer de valeurs qui sont de type *ValeurScalaire* (contrainte OCL 5.9).

```
context ImplantationComposant
    inv: self.SpécificationComposant → notEmpty()
    implies self. SpécificationComposant.Définition →
        select(d | d.oclIsTypeOf(DéfinitionPropriété)) →
            forAll(ds : DéfinitionPropriété | self.Définition →
                select(d | d.oclIsTypeOf(DéfinitionPropriété)).exists(d | d.id = ds.id))
context ImplantationComposant
    inv: self. Définition → select(d | d.oclIsTypeOf(DéfinitionPropriété)) →
        forAll(dp : DéfinitionPropriété | dp.Valeur → notEmpty()
            and dp.Valeur.oclIsTypeOf(ValeurScalaire))
```

Contrainte OCL 5.9: Définition de propriété au niveau implantation de composant.

Les composants ne disposant pas de propriétés, nous avons simplement la contrainte OCL 5.10.

```
context Composant
    inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionPropriété)).isEmpty()
```

Contrainte OCL 5.10: Absence de définition de propriété au niveau des composants.

La figure 5.16 présente un diagramme d'objets défini à partir de l'exemple donné à la figure 5.5 page 116.

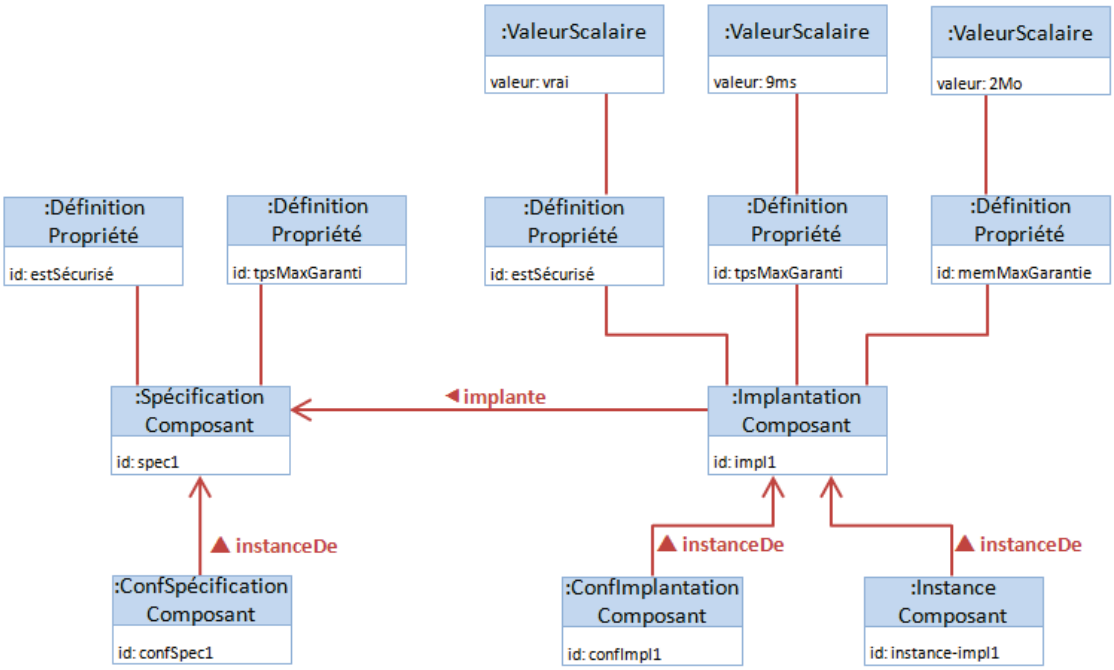

FIGURE 5.16 – Diagramme d'objets exprimant un ensemble de propriétés.

5.3.3 Expression des paramètres

Après avoir traité les propriétés et présenté l'ensemble des contraintes qui régissent leur utilisation, nous allons faire de même avec les paramètres. Pour illustrer notre proposition, nous allons nous baser sur l'exemple de la figure 5.17, qui reprend celui donné à la figure 5.6 page 117. De la même manière que précédemment, ce schéma nous montre une spécification de composant (*spec1*), une implantation de composant (*impl1*), une configuration de spécification de composant (*confSpec1*), une configuration d'implantation de composant (*confImpl1*) et une instance de composant (*instance-impl1*). Cette fois ci, l'accent est mis sur les définitions de paramètres et leurs valeurs associées.

Les spécifications de composants peuvent disposer d'un ensemble de définitions de paramètres, mais qui ne doivent pas être valuées (contrainte OCL 5.11) ; une valeur par défaut sera donnée par l'implantation. Dans notre exemple, *spec1* dispose de deux définitions de paramètres nommées *serverURL* et *nbThreads*.

context SpecificationComposant
  inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
    forAll(dp : DéfinitionParamètre | dp.Valeur → isEmpty())

Contrainte OCL 5.11: Définition de paramètre au niveau spécification de composant.

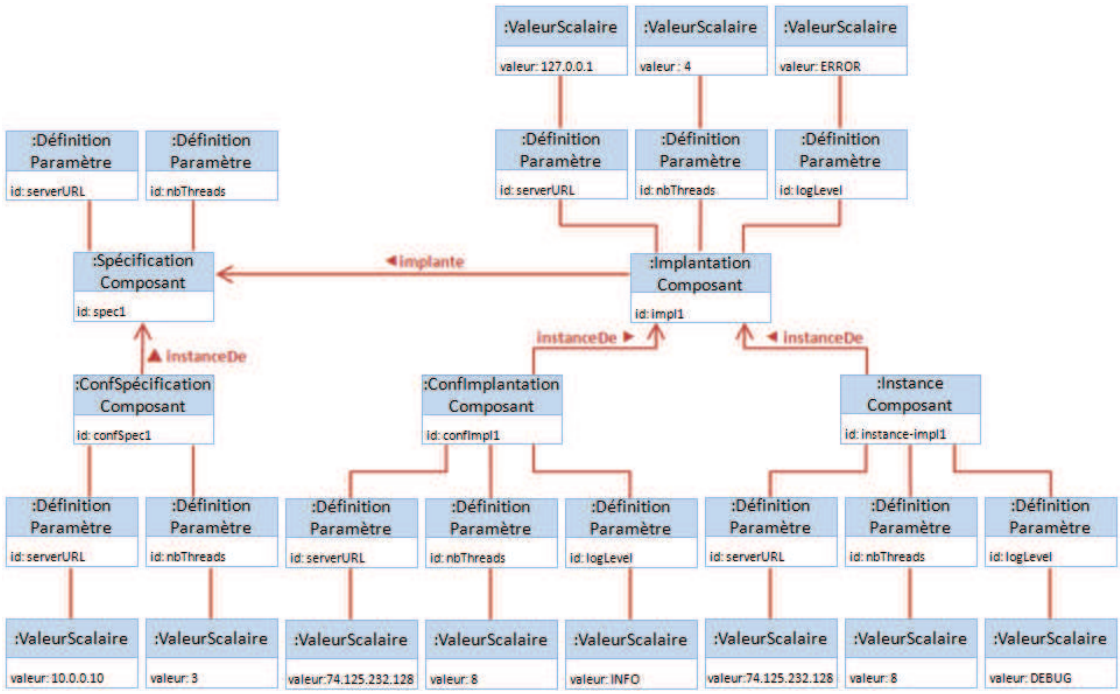

FIGURE 5.17 – Diagramme d'objets exprimant un ensemble de paramètres.

Les implantations peuvent avoir un nombre quelconque de paramètres, chacun d'eux disposant d'une valeur scalaire qui sera comprise comme étant la valeur par défaut de ce paramètre (contrainte OCL 5.12). De plus, si l'implantation implante une spécification, elle doit disposer des paramètres définis par cette dernière en leur attribuant aussi une valeur par défaut.

```
context ImplantationComposant
    inv: self.SpécificationComposant → notEmpty()
    implies self. SpécificationComposant.Définition →
        select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
            forAll(ds : DéfinitionParamètre | self.Définition →
                select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = ds.id))
context Context ImplantationComposant
    inv: self. Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
        forAll(dp : DéfinitionParamètre | dp.Valeur → notEmpty()
            and dp.Valeur.oclIsTypeOf(ValeurScalaire))
```

Contrainte OCL 5.12: Définition de paramètre au niveau implantation de composant.

Les configurations de spécifications de composants vont donner des valeurs scalaires aux définitions de paramètres des spécifications de composants. Elles ne peuvent ni ajouter, ni supprimer de paramètres (contrainte OCL 5.13). Notons que ces contraintes présupposent que toute configuration de spécification de composant instancie une spécification de com-

posant, comme nous l'avons exprimé dans la contrainte précédente.

```
context ConfSpécificationComposant
  inv: self.TypeComposant.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(ds : DéfinitionParamètre | self.Définition →
          select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = ds.id))
  inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(dself : DéfinitionParamètre | self. TypeComposant.Définition →
          select(d | d.oclIsTypeOf(DéfinitionParamètre)).
              exists(d | d.id = dself.id))
  inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(d : DéfinitionParamètre | d.Valeur →
          notEmpty and d.Valeur.oclIsTypeOf(ValeurScalaire))
```

Contrainte OCL 5.13: Définition de paramètre au niveau des configuration de spécifications.

Les contraintes associées aux configurations d'implantations de composants (contrainte OCL 5.14) sont proches de celles que nous venons d'écrire, la logique étant la même. La différence principale provient du fait qu'une configuration d'implantation de composant instancie une implantation de composant et non une spécification de composant comme précédemment.

```
context ConfImplantationComposant
  inv: self.TypeComposant.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(di : DéfinitionParamètre | self.Définition →
          select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = di.id))
  inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(dself : DéfinitionParamètre | self. TypeComposant.Définition →
          select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = dself.id))
  inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
      forAll(d : DéfinitionParamètre | d.Valeur →
          notEmpty and d.Valeur.oclIsTypeOf(ValeurScalaire))
```

Contrainte OCL 5.14: Définition de paramètre au niveau des configuration d'implantations.

Pour terminer avec les paramètres, il ne nous reste plus qu'à traiter ceux-ci au niveau des instances de composants. A l'exécution, la valeur des paramètres peut évoluer ; cependant, il n'est bien entendu ni possible d'en ajouter, ni d'en retirer par rapport à la liste définie dans leur implantation (contrainte OCL 5.15).

```
context InstanceComposant
    inv: self.TypeComposant.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
        forAll(di : DéfinitionParamètre | self.Définition →
            select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = di.id))
    inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
        forAll(dself : DéfinitionParamètre | self. TypeComposant.Définition →
            select(d | d.oclIsTypeOf(DéfinitionParamètre)).exists(d | d.id = dself.id))
    inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionParamètre)) →
        forAll(d : DéfinitionParamètre | d.Valeur →
            notEmpty and d.Valeur.oclIsTypeOf(ValeurScalaire))
```

Contrainte OCL 5.15: Définition de paramètre au niveau des instances de composant.

5.3.4 Expression des variables d'état

Comme les variables d'état sont destinées à représenter les phénomènes d'exécution, seules les instances peuvent disposer de variables d'état valuées. De plus, nous avons vu précédemment que les spécifications et les implantations de composants pouvaient disposer de définitions de variables d'état (contrainte OCL 5.16).

```
context SpécificationComposant
    inv: self.Définition → select(d.oclIsTypeOf(DéfinitionVariableEtat)) →
        forAll(d.Valeur → isEmpty())
context ImplantationComposant
    inv: self.Définition → select(d.oclIsTypeOf(DéfinitionVariableEtat)) →
        forAll(d.Valeur → isEmpty())
context ConfSpécificationComposant
    inv: self.Définition → select(d.oclIsTypeOf(DéfinitionVariableEtat)) → isEmpty()
context ConfImplantationComposant
    inv: self.Définition → select(d.oclIsTypeOf(DéfinitionVariableEtat)) → isEmpty()
context InstanceComposant
    inv: self.Définition → select(d.oclIsTypeOf(DéfinitionVariableEtat)) →
        forAll(d.Valeur.oclIsTypeOf(Collection))
```

Contrainte OCL 5.16: Localisation des variables d'état.

Dans l'exemple de la figure 5.18 exprimant ce qui a été présenté à la figure 5.7 page 118, nous retrouvons, comme dans les exemples précédents, une spécification de composant (*spec1*), une implantation de composant (*impl1*), une configuration de spécification de composant (*confSpec1*), une configuration d'implantation de composant (*confImpl1*) et, enfin, une instance de composant (*instance-impl1*).

La spécification de composant *spec1* dispose d'une définition de variable d'état qui a pour identifiant *tpsTraitementMoyen*. L'implantation de composant *impl1* doit posséder les mêmes définitions de variables d'état que la spécification qu'elle implante. Dans notre cas, elle a donc elle aussi une variable d'état qui a pour identifiant *tpsTraitementMoyen*. Elle peut

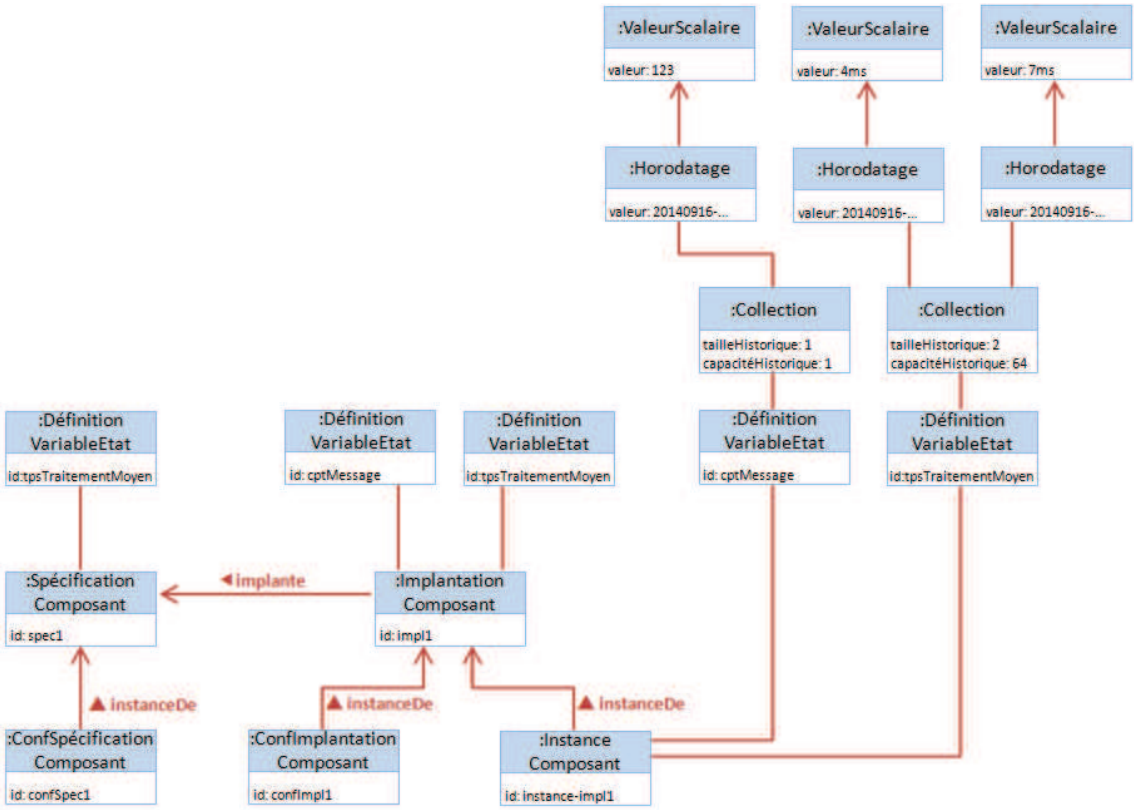

FIGURE 5.18 – Diagramme d'objets exprimant un ensemble de variables d'état.

aussi en ajouter d'autres, comme c'est le cas avec la variable d'état *cptMessage*. La formalisation est donnée par la contrainte OCL 5.17.

context ImplantationComposant
  inv: self.SpécificationComposant → notEmpty()
  implies self. SpécificationComposant.Définition →
    select(d | d.oclIsTypeOf(DéfinitionVariableEtat)) →
      forAll(ds : DéfinitionVariableEtat | self.Définition →
        select(d | d.oclIsTypeOf(DéfinitionVariableEtat)) → exists(d | d.id = ds.id))

Contrainte OCL 5.17: Héritage des variables d'état des spécifications aux implantations de composants.

Les instances de composants disposent des mêmes définitions que leur implantation (contrainte OCL 5.18). De plus, ces variables d'état disposent de valeurs qui sont de type collection. Elles permettent de suivre l'évolution de ces variables au cours du temps en conservant un historique quand cela est nécessaire. Dans notre exemple, il n'est pas nécessaire de disposer d'un historique pour un compteur de messages (sauf à vouloir garder une trace de la date d'arrivée des messages successifs). Par contre, il peut être intéressant de suivre l'évolution du temps moyen de traitement de chaque message. Les variables d'état *cptMes-*

sage et *tpsTraitementMoyen* disposent donc d'un historique de capacités respectives de 1 et 64 valeurs. Notons qu'à l'instant représenté par la figure 5.18, l'historique de la variable d'état *tpsTraitementMoyen* ne disposait que de deux valeurs horodatées.

```
context InstanceComposant
    inv: self.TypeComposant.Définition →
        select(d | d.oclIsTypeOf(DéfinitionVariableEtat)) →
            forAll(di : DéfinitionVariableEtat | self.Définition →
                select(d | d.oclIsTypeOf(DéfinitionVariableEtat)) → exists(d | d.id = di.id))
    inv: self.Définition → select(d | d.oclIsTypeOf(DéfinitionVariableEtat)) →
        forAll(dself : DéfinitionVariableEtat | self.TypeComposant.Définition →
            select(d | d.oclIsTypeOf(DéfinitionVariableEtat)).
                exists(d | d.id = dself.id))
```

Contrainte OCL 5.18: Identité des définitions de variables d'état entre une instance et son implantation.

5.3.5 Expression des contraintes

Pour terminer, nous nous intéressons aux contraintes. Nous avons vu qu'il était possible de définir des contraintes sur les propriétés, les paramètres et les variables d'état. Si les contraintes que nous allons énumérer couvrent l'ensemble de ces cas, l'exemple servant d'illustration sera celui de la figure 5.8 page 119 : une contrainte sur une propriété. Une représentation de l'instanciation correspondant à notre exemple se trouve à la figure 5.19. Nous avons une contrainte nommé *contrainteSécurité* qui est définie au niveau de la configuration de spécification *confSpec1*. Cette contrainte fait donc partie d'une architecture de conception, non représentée ici. Cela est cohérent avec le fait que toute contrainte est définie et valuée (par une valeur scalaire) uniquement dans ces architectures (contrainte OCL 5.19).

```
context TypeComposant
    inv: self. Définition → select(d | d.oclIsTypeOf(DéfinitionContrainte)) → isEmpty()
context Composant
    inv: self. Définition → select(d | d.oclIsTypeOf(DéfinitionContrainte)) → notEmpty()
    implies self.Architecture.oclIsTypeOf(ArchitectureConception)
context DéfinitionContrainte
    inv: self.Valeur → notEmpty() and self.Valeur.oclIsTypeOf(ValeurScalaire)
```

Contrainte OCL 5.19: Définitions et valeurs des contraintes.

Dans notre exemple, nous pouvons remarquer que la définition de propriété *estSécurisée*, qui est référencée par la contrainte ne dispose pas de valeur. Cela est tout à fait normal, puisque cette dernière est attachée directement au niveau de la contrainte elle-même (contrainte OCL 5.20).

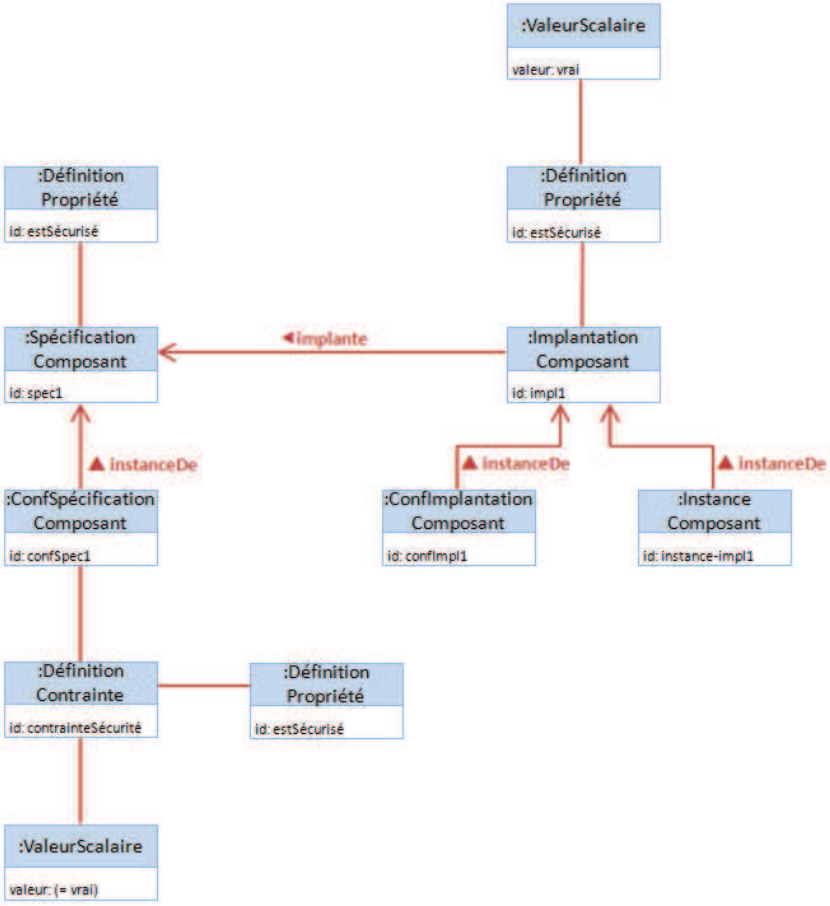

FIGURE 5.19 – Diagramme d'objets exprimant une contrainte.

context DéfinitionContrainte
inv: self.Définition.Valeur → isEmpty()

Contrainte OCL 5.20: Absence de valeur dans la définition de contrainte.

Enfin, pour garantir une effectivité de la contrainte, nous souhaitons imposer que le *TypeComposant* instancié par le *Composant* qui dispose de la contrainte ait la définition objet de la contrainte. Dans notre exemple, nous avons une contrainte sur la définition de propriété *estSécurisée*. Il faut donc que le type composant *spec1* instancié par le composant *confSpec1* qui porte la contrainte *contrainteSécurité* dispose lui aussi de la définition de propriété *estSécurisé*, ce qui est effectivement le cas. Cette dernière règle est formalisée par la contrainte OCL 5.21.

context DéfinitionContrainte
inv: self.Composant.TypeComposant.Definition →
exists(d | d.id = self.Definition.id and d.oclIsTypeOf(self.Definition))

Contrainte OCL 5.21: Cohérence avec le type instancié par l'objet qui porte la contrainte.

5.4 Validation des architectures

5.4.1 Motivations et démarche

Pour structurer la connaissance nécessaire à l'administration des plates-formes à composants orientés services dynamiques tout au long de leur cycle de vie, notre proposition prend appui sur les architectures. Nous les avons précédemment formalisées. Nous avons défini deux types de transformation dans le chapitre 4 (paragraphe 4.2.3 page 103), illustrés par la figure 5.20. La première transformation ❶ entre l'architecture de conception et l'architecture de déploiement est une transformation typique qui correspond à la définition d'un plan de déploiement. La deuxième transformation entre l'architecture de déploiement et l'architecture de l'exécution est composée de deux étapes : l'instanciation ❷ du modèle de déploiement, puis la réification ❸ de l'architecture exécutée en une architecture de l'exécution. La réification tient compte des différentes évolutions des élements de la plate-forme au cours du temps. L'architecture de l'exécution évolue donc selon l'arrivée et le départ de certains composants ; de plus, les liaisons entre composants peuvent également évoluer. Cette plasticité de l'architecture de l'exécution implique de mettre en place un mécanisme de vérification ❹ de la confirmité de ce qui est exécuté par rapport à ce qui a été conçu.

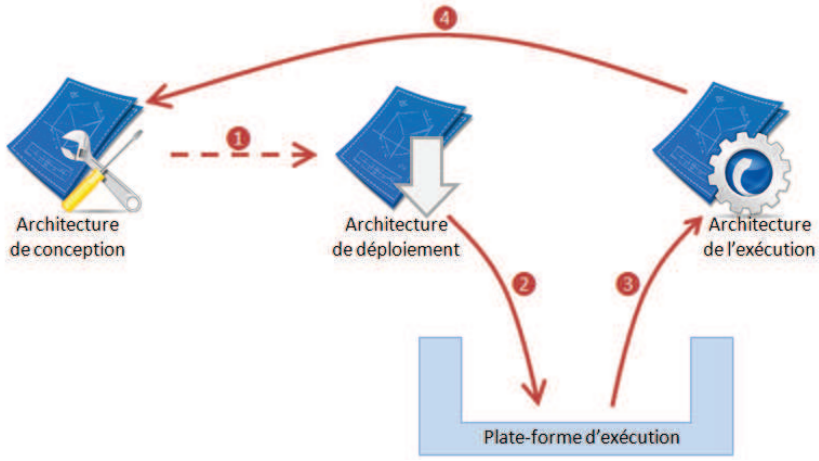

FIGURE 5.20 – Les transformations de modèles.

Dans cette partie, nous nous intéressons à la vérification de la conformité de l'architecture de l'exécution avec celle de conception. Plus précisément, l'objectif final n'est pas seulement d'obtenir une réponse binaire quant au respect des contraintes de conception, mais plus que cela, **de disposer d'un ensemble de points localisés où l'architecture de l'exécution est en conflit avec l'architecture de conception, afin de faciliter le processus de mise en conformité**.

Nous proposons pour cela une **validation en deux étapes**. La première est effectuée par l'**algorithme de correspondance**. Il cherche à déterminer pour chaque composant de l'architecture de l'exécution le composant de l'architecture de conception auquel il correspond (figure 5.21).

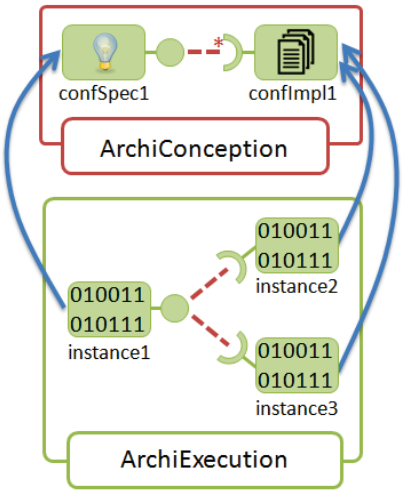

FIGURE 5.21 – Illustration du résultat souhaité par l'algorithme de correspondance.

Dans une seconde étape, la correspondance exacte entre les composants des deux architectures est vérifiée par un algorithme appelé **algorithme de vérification**. Les erreurs de granularité fine (comme une violation de contrainte sur les propriétés d'un composant ou sur la cardinalité d'un lien) sont alors détectées.

Pour présenter le fonctionnement de ces algorithmes, nous allons commencer par définir les objets mathématiques dont nous aurons besoin (paragraphe 5.4.2). Ensuite, nous détaillerons l'algorithme de correspondance (paragraphe 5.4.3 page 142) et l'algorithme de vérification (paragraphe 5.4.4 page 149). Notons que notre algorithme de correspondance a été conçu pour permettre un calcul incrémental, ce qui permet de capitaliser sur les résultats des exécutions précédentes de l'algorithme de correspondance. Pour ce qui est de la seconde étape (algorithme de vérification), la faible complexité des calculs rend inutile de s'appuyer sur les résultats des vérifications précédentes.

5.4.2 Définition des objets mathématiques utilisés

Pour commencer, définissons $\mathcal{B} = \{vrai, faux\}$, l'ensemble des valeurs booléennes.

Considérons les composants :

- soit C_{spec} l'ensemble des spécifications de composants ;
- soit C_{impl} l'ensemble des implantations de composants ;
- soit $C_{type} = C_{spec} \cup C_{impl}$, l'ensemble des types de composants ;
- soit $C_{confspec}$ l'ensemble des configurations de spécification de composants ;
- soit $C_{confimpl}$ l'ensemble des configurations d'implantation de composants ;
- soit C_{inst} l'ensemble des instances de composants ;
- soit $C_{cmp} = C_{confspec} \cup C_{confimpl} \cup C_{inst}$, l'ensemble des composants ;

- soit id, une fonction qui a un composant (type ou objet) fait correspondre son identifiant sous forme d'une chaîne de caractères.

Soit $spécification$ une fonction qui fait correspondre pour chaque implantation de composant sa spécification s'il en a une, et l'ensemble vide dans le cas contraire :

$$spécification : C_{impl} \longrightarrow C_{spec} \cup \{\varnothing\}$$

Soit $estImplantation$ (respectivement, $estSpécification$) une fonction qui fait correspondre un booléen à un type composant selon que ce type composant est ou non une implantation de composant (respectivement, une spécification) :

$$\begin{aligned} estImplantation : C_{type} &\longrightarrow \mathcal{B} \\ estSpécification : C_{type} &\longrightarrow \mathcal{B} \end{aligned}$$

Soit $type$, une fonction qui fait correspondre à un objet composant le type composant dont il est instance. Cette fonction est définie pour tout composant $c \in C_{cmp}$:

$$\begin{aligned} type : C_{confspec} &\longrightarrow C_{spec} \\ type : C_{confimpl} &\longrightarrow C_{impl} \\ type : C_{inst} &\longrightarrow C_{impl} \end{aligned}$$

Formalisons maintenant les interfaces :

- soit I_f l'ensemble des interfaces de services fournies de tous les composants ;
- soit I_r l'ensemble des interfaces de services requises de tous les composants ;
- soit $I = I_f \cup I_r$, l'ensemble des interfaces de services ;
- soit CS l'ensemble de tous les contrats de services ;
- soit $contratService$ une fonction qui fait correspondre à une interface (requise ou fournie) le contrat de service qui lui est associé :

$$contratService : I \longrightarrow CS$$

Soit $interfacesFournies$ (respectivement, $interfacesRequises$), une fonction qui fait correspondre à un type composant l'ensemble de ses interfaces fournies (respectivement, requises). Cette fonction est définie pour tout composant $c \in C_{type}$:

$$\begin{aligned} interfacesFournies : C_{type} &\longrightarrow \mathcal{P}(I_f) \\ interfacesRequises : C_{type} &\longrightarrow \mathcal{P}(I_r) \end{aligned}$$

Passons maintenant aux liaisons :

- soit L_{con} l'ensemble des liaisons de conception, qui peuvent donc présenter de la variabilité ;
- soit L_{exec} l'ensemble des liaisons à l'exécution, sans variabilité.

Soit $fournisseur$ (respectivement, $consommateur$), une fonction qui fait correspondre à une liaison le composant qui est en lien avec cette liaison via une interface fournie (respectivement, requise). Cette fonction est définie pour toute liaison $l \in L_{con} \cup L_{exec}$:

$$\begin{aligned} fournisseur : L_{con} &\longrightarrow C_{cmp} \\ fournisseur : L_{exec} &\longrightarrow C_{cmp} \\ consommateur : L_{con} &\longrightarrow C_{cmp} \\ consommateur : L_{exec} &\longrightarrow C_{cmp} \end{aligned}$$

Nous étendons la fonction $contratService$ pour faire correspondre à une liaison le contrat de service qui lui est associé :

$$contratService : L_{con} \cup L_{exec} \longrightarrow CS$$

Définissons à présent les architectures. Soit $\mathcal{A}_c$ l'ensemble des architectures de conception et $\mathcal{A}_e$ l'ensemble des architectures de l'exécution. Nous avons :

$$\begin{aligned} A_c = (C_c, L_c) \in \mathcal{A}_c &\Rightarrow C_c \subseteq C_{confspec} \cup C_{confimpl} \wedge L_c \subseteq L_{con} \\ A_e = (C_e, L_e) \in \mathcal{A}_e &\Rightarrow C_e \subseteq C_{inst} \wedge L_e \subseteq L_{exec} \end{aligned}$$

Nous définissons maintenant la fonction $estCompatible$. Un composant c_1 d'une architecture de conception et un composant c_2 d'une architecture de l'exécution sont dits compatibles s'ils sont tous les deux instance du même type composant ou si la spécification (si elle existe) du type de c_2 est égale au type de c_1 :

$$\begin{gathered} estCompatible : (C_{confspec} \cup C_{confimpl}) \times C_{inst} \longrightarrow \mathcal{B} \\ (c_1, c_2) \longmapsto type(c_1) = type(c_2) \vee type(c_1) = spécification(type(c_2)) \end{gathered}$$

Nous faisons à présent de même avec les liens, en définissant une nouvelle fonction $estCompatible$. Une liaison issue d'une architecture de conception et une liaison issue d'une architecture de l'exécution sont compatibles si leurs fournisseurs et leurs consommateurs sont compatibles. De plus, il est nécessaire que le contrat de services des interfaces des deux liaisons soit identique :

$$estCompatible : L_{con} \times L_{exec} \longrightarrow \mathcal{B}$$

$$(l_1, l_2) \longmapsto \begin{cases} faux & \text{si } contratService(l_1) \neq contratService(l_2) \\ faux & \text{si } estCompatible(fournisseur(l_1), fournisseur(l_2)) = faux \\ faux & \text{si } estCompatible(consommateur(l_1), consommateur(l_2)) = faux \\ vrai & \text{sinon} \end{cases}$$

Intéressons nous à présent aux contraintes. Nous définissons une fonction $contraintes$ qui retourne l'ensemble des définitions de contraintes d'un composant. De plus, nous définissons une fonction $contrainteSatisfaite(dc, cmp)$ qui permet de tester si un composant cmp satisfait une définition de contrainte dc (nécessairement valuée).

5.4.3 Algorithme de correspondance

5.4.3.1 Présentation générale

L'algorithme de correspondance précède l'algorithme de vérification que nous présenterons dans la paragraphe 5.4.4 page 149. A eux deux, ils permettent de valider une architecture de l'exécution au regard de son architecture de conception.

Etant données une architecture de conception $A_c = (C_c, L_c) \in \mathcal{A}_c$ et une architecture de l'exécution $A_e = (C_e, L_e) \in \mathcal{A}_e$, l'objectif de l'algorithme de correspondance est de définir un tableau associatif $correspondance$ qui, à une instance de composant $i \in C_e$, associe $correspondance[i] \in C_c$, le composant de l'architecture de conception qui justifie la présence du composant i dans l'architecture de l'exécution. Pour cela, l'algorithme de correspondance est structuré selon une étape d'initialisation et une série d'itérations.

L'étape d'initialisation a une double tâche :

- **le calcul de tableaux de valeurs :** ceux-ci étant utilisés lors de la seconde étape de l'initialisation et à chaque itération, ces tableaux seront calculés en début de traitement ;
- **l'initialisation du tableau associatif** $correspondance$ **:** pour cela, l'étape d'initialisation peut réutiliser les résultats d'une précédente exécution de l'algorithme de correspondance en filtrant les valeurs pour ne garder que celles pertinentes en fonction du nouveau contexte d'exécution. Cela permet de ne calculer que les correspondances pour les composants ajoutés entre temps à l'architecture d'exécution.

A la suite de cette initialisation, une série d'itérations est effectuée. Les itérations sont paramétrées par une fonction de correspondance, qui prend en paramètre une instance de l'architecture de l'exécution et retourne le composant qui lui correspond dans l'architecture de conception (ou NIL, s'il n'est pas parvenu à le déterminer). C'est cette **fonction de correspondance** qui détermine la précision de l'itération. La première itération recherche des correspondances extrêmement précises et donc extrêmement fiables du point de vue du respect des contraintes de conception. Les suivantes sont de moins en moins exigeantes de ce

point de vue, ce qui permet de détecter en seconde intention des correspondances pour des éléments qui ne respectent pas l'architecture de conception. Nous rappelons qu'à la suite de l'algorithme de correspondance, le respect des contraintes de conception est vérifié, ce qui nous permet de disposer d'un ensemble de points de correction où celle-ci est violée, le cas échéant.

Le fonctionnement général d'une itération est le suivant. Pour chaque instance de l'architecture de l'exécution dont l'algorithme n'a pas encore trouvé la correspondance, on applique la fonction de correspondance. Si celle-ci découvre une nouvelle correspondance, l'algorithme s'appuie sur les liaisons pour trouver d'autres correspondances. Pour expliquer le fonctionnement de ce mécanisme, nous allons commenter la figure 5.22.

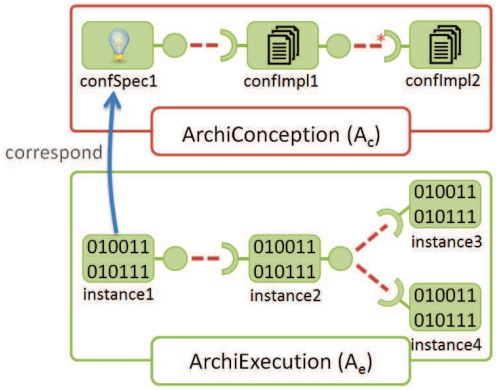

FIGURE 5.22 – Illustration du parcours des liaisons pour le calcul des correspondances.

Nous supposons que la fonction de correspondance a permis d'établir que la correspondance de *instance1* est la configuration de spécification *confSpec1* de l'architecture de conception A_c. Comme *confSpec1* a une unique liaison dont il est fournisseur, on peut sans aucun doute établir une correspondance entre *instance2* et *confImpl1*. En s'appuyant sur cette nouvelle correspondance et en constatant que *confImpl1* a lui aussi une unique liaison dont il est le fournisseur, on peut déduire que *instance3* et *instance 4* ont pour correspondance *confImpl2*. Nous affinerons le fonctionnement de l'algorithme en le présentant en détails par la suite.

Nous allons maintenant donner l'algorithme de correspondance complet (algorithme 5.1). Comme nous l'avons dit, cet algorithme doit supporter la vérification incrémentale des architectures. C'est la raison pour laquelle le paramètre correspondance est en entrée/sortie. Dans le cas où l'on désire une validation totale de l'architecture, il suffira d'initialiser la variable correspondance de la manière suivante : $\forall c \in C_e, correspondance[c] = NIL$ Dans le cas contraire, la précédente valeur du tableau correspondance devra être fournie. Notons enfin que l'algorithme retourne aussi un booléen ($correspondanceComplète$) qui permet de savoir si $\forall c \in C_e, correspondance[c] \neq NIL$.

Algorithme 5.1 Algorithme de correspondance.

Entrée : $A_c = (C_c, L_c)$, une architecture de conception
Entrée : $A_e = (C_e, L_e)$, une architecture de l'exécution
Entrée/Sortie : $correspondance$, le tableau associatif de correspondances
Sortie : $correspondanceComplète$, un booléen

1: ▷ initialisations
2: tabs ← initialisationTableaux(A_c,A_e)
3: correspondance ← initialisationCorrespondance(A_c,A_e,correspondance)
4: **si** $\forall c \in C_e, correspondance[c] \neq NIL$ **alors**
5: correspondanceComplète ← vrai
6: **retourner** (correspondance, correspondanceComplète)
7: **fin condition**
8:
9: ▷ itérations
10: fn-corresp ← {correspondance-compatibilité, correspondance-compatibilité-id, correspondance-id}
11: **pour** f ∈ fn-corresp **faire**
12: iteration(A_c, A_e, correspondance, f, tabs)
13: **si** $\forall c \in C_e, correspondance[c] \neq NIL$ **alors**
14: correspondanceComplète ← vrai
15: **retourner** (correspondance, correspondanceComplète)
16: **fin condition**
17: **fin pour**
18:
19: correspondanceComplète ← faux
20: **retourner** (correspondance, correspondanceComplète)

Maintenant que nous avons présenté le fonctionnement général de l'algorithme de correspondance, nous allons donner les fonctions de correspondance qui permettent de paramétrer les itérations.

ITÉRATION 1 : fonction de correspondance avec la compatibilité seule

Lors de la première itération, la fonction de correspondance regarde la compatibilité entre l'instance considérée et les éléments de l'architecture de conception. Si un et un seul composant de l'architecture de conception est compatible avec l'instance, une relation de correspondance peut être établie avec celui-ci.

La fonction $correspondance\text{-}compatibilité(instance, A_c = (C_c, L_c), A_e = (C_e, L_e), tabs)$ est définie de la manière suivante :

$$\begin{cases} c \in tabs.composantsCompatibles[instance], \text{ si } |tabs.composantsCompatibles[instance]| = 1 \\ NIL \text{ sinon} \end{cases}$$

ITÉRATION 2 : fonction de correspondance avec la compatibilité et l'inclusion des noms

L'objectif de cette seconde fonction de correspondance est de tenter d'établir une correspondance dans le cas où une instance dispose de plusieurs objets compatibles dans l'architecture de conception. Si un et un seul composant de l'architecture de conception compatible avec l'instance dispose d'un identifiant inclus dans l'identifiant de l'instance, alors une relation de correspondance peut être établie avec celui-ci. En effet, dans la pratique, il n'est pas rare que les identifiants des instances soient formés à partir des identifiants de leur correspondance dans l'architecture de conception.

La fonction $correspondance\text{-}compatibilité\text{-}id(instance, A_c = (C_c, L_c), A_e = (C_e, L_e), tabs)$ est définie de la manière suivante :

$$\begin{cases} c & \text{si } \exists! c \in tabs.composantsCompatibles[instance] | id(c) \subseteq id(instance) \\ NIL & \text{sinon} \end{cases}$$

ITÉRATION 3 : fonction de correspondance avec seulement l'inclusion des noms

Cette dernière fonction de correspondance, qui est utilisée lors de la troisième itération ne se base que sur les identifiants. Ainsi, il est éventuellement possible d'établir une correspondance entre deux composants non compatibles. De cette manière, lors de la vérification qui suit, une erreur sur le type de composant peut être détectée.

La fonction $correspondance\text{-}id(instance, A_c = (C_c, L_c), A_e = (C_e, L_e), tabs)$ est définie de la manière suivante :

$$\begin{cases} c & \text{si } \exists! c \in C_c | id(c) \subseteq id(instance) \\ NIL & \text{sinon} \end{cases}$$

Maintenant que nous avons présenté l'algorithme de correspondance dans ses grandes lignes, nous allons successivement décrire les initialisations et l'algorithme des itérations.

5.4.3.2 Les initialisations

Pour éviter des parcours répétitifs des architectures, les tableaux de listes suivants sont calculés avant les itérations pour être fournis à chacune d'elles.

- $\forall c \in C_c, liaisonsConceptionFournisseur[c] = \{l \in L_c | fournisseur(l) = c\}$
- $\forall c \in C_c, liaisonsConceptionConsommateur[c] = \{l \in L_c | consommateur(l) = c\}$
- $\forall c \in C_e, liaisonsExcutionFournisseur[c] = \{l \in L_c | fournisseur(l) = c\}$
- $\forall c \in C_e, liaisonsExcutionConsommateur[c] = \{l \in L_c | consommateur(l) = c\}$
- $\forall i \in C_e, composantsCompatibles[i] = \{c \in C_c | estCompatible(c, i) = vrai\}$

Ces tableaux sont regroupés dans une structure (nommé $tabs$). L'accès aux tableaux se fera en consultant les champs de la structure. Ainsi, $tabs.liaisonsConceptionFournisseur$ retournera le tableau associatif $liaisonsConceptionFournisseur$ précédemment défini.

La seconde tâche de l'initialisation est la préparation du tableau $correspondance$. Pour cela, on vérifie que le tableau dispose d'entrées pour tous les composants de l'architecture exécutée : $\forall c \in C_e$, si $correspondance[c] = UNDEFINED, correspondance[c] \leftarrow NIL$. Symétriquement, on supprime les entrées dans le tableau associatif correspondance pour $\forall c \notin C_e$

L'initialisation de ces tableaux est effectuée en parcourant séquenciellement l'ensemble des liaisons de l'architecture de conception (pour les quatre premiers tableaux) et des composants de l'architecture de l'exécution pour le dernier tableau associatif.

5.4.3.3 Les itérations

Algorithme principal

L'objectif de chaque itération de l'algorithme est de trouver des correspondances entre des composants de l'architecture exécutée $A_e = (C_e, L_e)$ et des composants de l'architecture de conception $A_c = (C_c, L_c)$. Pour conserver cette information, l'algorithme dispose en entrée/sortie d'une variable $correspondance$. Celle-ci est un tableau associatif défini pour chaque composant de l'architecture de l'exécution $c \in C_e$.

Algorithme 5.2 Algorithme principal des itérations.

Entrée : $A_c = (C_c, L_c)$, une architecture de conception
Entrée : $A_e = (C_e, L_e)$, une architecture de l'exécution
Entrée : $fonctionDeCorrespondance$
Entrée : $tabs$, une structure contenant un ensemble de tableaux associatifs calculés à l'initialisation
Entrée/Sortie : $correspondance$, le tableau associatif de correspondance

1: aTester ← {instance ∈ C_e | correspondance[instance] = NIL}
2: **tant que** |aTester| ≠ 0 **faire**
3: sélectionner instance ∈ aTester
4: aTester ← aTester \ instance
5: element ← fonctionDeCorrespondance(instance, A_c, A_e, tabs)
6: **si** element ≠ NIL **alors**
7: correspondance[instance] ← element
8: explorationLiensFournisseur(A_c, A_e, correspondance, aTester, instance, element, tabs)
9: explorationLiensConsommateur(A_c, A_e, correspondance, aTester, instance, element, tabs)
10: **fin condition**
11: **fin tant que**

L'algorithme est aussi paramétré à l'aide d'une fonction de correspondance. Celle-ci a pour objectif de trouver pour un composant de l'architecture de l'exécution le composant de

l'architecture de conception auquel il fait référence. Cette fonction retourne donc cet élément ou NIL si elle ne le trouve pas.

L'exécution de l'algorithme 5.2 se déroule de la manière suivante. On initialise la liste $aTester$ avec l'ensemble des instances de l'architecture de l'exécution dont on ne connaît pas encore la correspondance. Ensuite, on itère tant qu'il reste des éléments dans cette liste. Notons que nous ne faisons pas un parcours linéaire de la liste, car nous verrons qu'elle peut être modifiée lors de l'exploration des liens fournisseurs et consommateurs.

Tant que la liste $aTester$ n'est pas vide, on retire un élément de celle-ci et on regarde si l'on trouve une correspondance pour celui-ci à l'aide de la fonction passée en paramètre. Dans le cas positif, cette correspondance est enregistrée dans la variable éponyme et on lance l'exploration des liens fournisseurs et consommateurs. Lors de celle-ci, de nouvelles correspondances peuvent être trouvées. Dans ce cas, les fonctions d'exploration suppriment les instances correspondantes de la variable $aTester$ et elles mettent à jour la variable $correspondance$ (algorithme 5.3).

Les deux cas les plus défavorables lors de l'exécution de l'algorithme sont les suivants :

- la fonction de correspondance renvoie toujours NIL : dans ce cas, elle est appelée (sans succès) pour chaque élément de la liste $aTester$ et il faudra exécuter les itérations suivantes ;
- la fonction de correspondance retourne toujours un résultat différent de NIL : l'exploration des liens fournisseur et consommateur peut alors ajouter des éléments à la liste $aTester$. Dans le cas le plus défavorable, tous les composants de l'architecure de l'exécution sont ajoutés. Cependant, ces ajouts sont la conséquence de correspondances trouvées ; celles-ci ne devront donc plus être recherchées aux itérations suivantes.

Algorithme d'exploration des liens fournisseur

L'algorithme d'exploration des liens fournisseur (algorithme 5.3) va s'appuyer sur une correspondance entre une instance ($instanceInit$) et un élément de l'architecture de conception ($elementInit$) afin de découvrir de nouvelles correspondances. Lors de l'exécution, les variables $aTester$ et $correspondance$ fournies en paramètre seront mises à jour si de nouvelles correspondances sont trouvées.

L'exploration des liaisons peut conduire à un parcours arborescent de l'architecture de l'exécution. En effet, d'une correspondance, on peut potentiellement en découvrir plusieurs autres. Par exemple, dans le cas présenté à la figure 5.22 page 143, la correspondance *instance1/confSpec1* permettait de trouver une seule autre correspondance *instance2/confImpl2*. Cette seconde correspondance permettait, quant à elle, de découvrir non pas une mais deux autres correspondances (*instance3/confImpl2* et *instance4/confImpl2*) conduisant ainsi à un parcours d'arbre. Pour enregistrer les correspondances restant à explorer, nous utilisons la liste *aVisiter*. Celle-ci est initialisée avec une paire ordonnée crée à l'aide de la correspondance fournie en paramètre. Nous retirerons les correspondances pour les traiter une à une. Chaque correspondance trouvée est ajoutée à la liste pour être explorée ultérieurement. Les liaisons

n'étant parcourues qu'une seule fois, l'algorithme va explorer au plus l'ensemble des liaisons de l'architecture de l'exécution.

Algorithme 5.3 Exploration des liens fournisseur.

Entrée : $A_c = (C_c, L_c)$, une architecture de conception
Entrée : $A_e = (C_e, L_e)$, une architecture de l'exécution
Entrée : $instanceInit$
Entrée : $elementInit$
Entrée : $tabs$, une structure contenant un ensemble de tableaux associatifs calculés à l'initialisation
Entrée/Sortie : $aTester$
Entrée/Sortie : $correspondance$, le tableau associatif de correspondance

1: ▷ initialisation
2: aVisiter ← {(instanceInit, elementInit)}
3:
4: **tant que** |aVisiter| ≠ 0 **faire**
5: sélectionner (instance, element) ∈ aVisiter
6: aVisiter ← aVisiter \ (instance, element)
7: **pour tout** ic ∈ interfacesFournies(type(element)) **faire**
8: **si** ∃! lc ∈ tabs.liaisonsConceptionFournisseur[element] | lc.idItfFournie = ic.id **alors**
9: dst ← consommateur(lc)
10: **pour tout** le ∈ tabs.liaisonsExecutionFournisseur[instance] | le.idItfFournie = ic.id **faire**
11: src ← consommateur(le)
12: ▷ si nous venons de découvrir une nouvelle correspondance
13: **si** correspondance[src] = NIL **alors**
14: correspondance[src] ← dst
15: aTester ← aTester \ {src}
16: aVisiter ← aVisiter ∪ {(src, dst)}
17: **fin condition**
18: **fin pour**
19: **fin condition**
20: **fin pour**
21: **fin tant que**

Le suivi des liaisons utilise les identifiants des interfaces. De cette manière, il est possible de distinguer celles-ci au sein d'un même composant et d'éviter des erreurs de correspondance dans le cas où l'architecture de conception n'a pas été respectée.

Algorithme d'exploration des liens consommateur

L'exploration des liens côté consommateur est absolument symétrique à celle côté fournisseur. Cet algorithme d'exploration des liens consommateurs est donc tout à fait analogue à l'algorithme 5.3 détaillé au paragraphe précédent.

5.4.4 Algorithme de vérification

Comme nous l'avons dit, la validation d'une architecture exécutée au regard de son architecture de conception est effectuée par deux algorithmes successifs : l'algorithme de correspondance et l'algorithme de vérification.

Dans le paragraphe précédent, nous avons détaillé l'algorithme de correspondance qui permet pour un couple formé par une architecture de conception $A_c = (C_c, L_c) \in \mathcal{A}_c$ et une architecture de l'exécution $A_e = (C_e, L_e) \in \mathcal{A}_e$ de définir un tableau associatif $correspondance$ qui à une instance i de l'architecture de l'exécution fait correspondre le composant de l'architecture de conception qui justifie sa présence dans l'architecture de l'exécution.

En nous appuyant sur le tableau associatif $correspondance$, nous allons à présent donner les fonctions de vérification de la conformité entre les architectures de conception et de l'exécution. Pour simplifier la lecture, nous avons réparti celles-ci en deux catégories : les fonctions qui vérifient les composants et les fonctions qui vérifient les liens.

5.4.4.1 Vérification des composants

Définition de la variable correspondance

Tout d'abord, il convient de vérifier que chaque élément de l'architecture de l'exécution dispose d'une correspondance dans l'architecture de conception. Cela est effectué en retournant une erreur pour chaque composant $c \in C_e | correspondance[c] = NIL$.

En effet, il est possible que certains éléments de l'architecture de l'exécution ne puissent être mis en correspondance avec des éléments de l'architecture de conception. Cela peut avoir plusieurs causes :

- il y a une erreur dans le modèle de l'exécution et il est normal de ne pas obtenir de lien de correspondance pour un élément ;
- une ambiguïté demeure et il n'est pas possible de déterminer la correspondance entre les éléments de l'architecture de conception et ceux de l'architecture de l'exécution. En effet, le calcul de la correspondance est un problème qui peut ne pas être décidable.

Dans tous les cas, une erreur est reportée.

Compatibilité

Une seconde vérification va regarder la compatibilité entre les composants mis en relation par la variable $correspondance$. En d'autres termes, $\forall c \in C_e | correspondance[c] \neq NIL$, si $estCompatible(correspondance[c], c) = faux$, il faut retourner une erreur indiquant que le type de l'instance n'est pas valide.

Contraintes

La dernière vérification que nous souhaitons effectuer au niveau des composants concerne les contraintes. Comme nous l'avons vu, il est possible de définir des contraintes sur les propriétés, paramètres et variables d'état.

Nous allons donc vérifier que $\forall c_e \in C_e | correspondance[c_e] \neq NIL$ et pour chaque contrainte $c = contraintes(correspondance[c_e])$, $contrainteSatisfaite(c, c_e) = vrai$. Dans le cas contraire, une erreur est reportée pour chaque contrainte violée.

5.4.4.2 Vérification des liaisons

Pour la vérification des liaisons, nous allons commencer par définir deux tableaux associatifs que nous exploiterons par la suite.

Le premier est $correspondanceLiaison$. De la même manière que $correspondance[c]$ permet de déterminer le composant de l'architecture de conception $A_c = (C_c, L_c)$ qui correspond à un composant de l'architecture d'exécution $A_e = (C_e, L_e)$, $correspondanceLiaison[l]$ fait de même avec les liaisons :

$$\forall l \in L_e, correspondanceLiaison[l] =$$
$$\begin{cases} lc \in L_c & \text{si } correspondance[fournisseur(l)] = fournisseur(lc) \\ & \wedge correspondance[consommateur(l)] = consommateur(lc) \\ & \wedge estCompatible(lc, l) = vrai \\ NIL & \text{sinon} \end{cases}$$

Le second tableau associatif, nommé $correspondanceLiaisonInv$, permet d'obtenir dans l'architecture d'exécution $A_e = (C_e, L_e)$ l'ensemble des liaisons l dont une liaison lc de l'architecture de conception $A_c = (C_c, L_c)$ est la correspondante. Notons que contrairement à $correspondanceLiaison$, $correspondanceLiaisonInv$ est un tableau associatif de *listes* de liaisons :

$$\forall lc \in L_c, correspondanceLiaisonInv[lc] = \{l \in L_e | lc = correspondanceLiaison[l]\}$$

Le calcul des variables $correspondanceLiaison$ et $correspondanceLiaisonInv$ est effectué simultanément dans l'implantation en parcourant de manière séquencielle l'ensemble des liaisons de l'architecture de l'exécution.

A présent, nous pouvons vérifier la conformité des liaisons. Pour cela, nous commençons par vérifier que toutes les liaisons de l'architecture de l'exécution ont une correspondance dans l'architecture de conception ; c'est-à-dire que $\forall l \in L_e, correspondanceLiaison[l] \neq NIL$. Dans le cas contraire, la liaison l de l'architecture de l'exécution ne devrait pas exister et une erreur est reportée.

Il ne reste plus qu'à vérifier le respect des cardinalités en sélectionnant tout d'abord l'ensemble des liaisons à vérifier dans l'architecture de conception. Pour cela, nous calculons tout d'abord l'ensemble des correspondants définis de la manière suivante :

$$correspondants = \left(\bigcup_{c \in C_e} correspondance[c] \right) \setminus \{NIL\}$$

Les liaisons de l'architecture de conception qu'il faut vérifier sont celles dont le fournisseur ou le consommateur appartient à l'ensemble *correspondants* ; c'est-à-dire :

$$\{l \in L_c | fournisseur(l) \in correspondants \vee consommateur(l) \in correspondants\}$$

Pour chaque liaison de l'architecture de conception qu'il faut vérifier, nous appliquons l'algorithme 5.4. Pour illustrer son fonctionnement, nous allons nous appuyer sur l'exemple de la figure 5.23. Sur celle-ci, nous allons vérifier le respect des cardinalités de l'unique lien de l'architecture de conception que nous nommons ici *liaison*. Nous considérons successivement les cardinalités côté fournisseur puis consommateur de service de *liaison*.

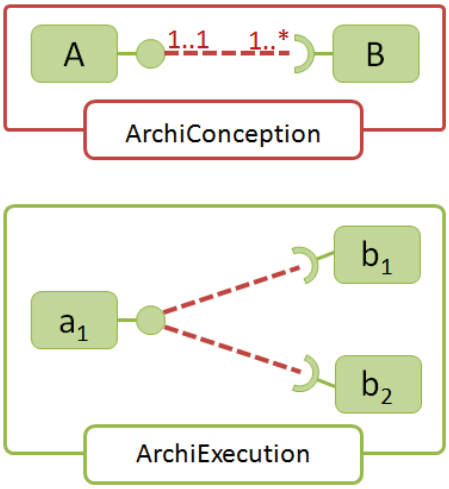

FIGURE 5.23 – Exemple pour l'explication de l'algorithme verification-cardinalité-liaison.

Pour le calcul côté fournisseur, nous calculons tout d'abord *liste-fournisseurs*, l'ensemble des fournisseurs de service des liaisons appartenant à la liste $correspondanceLiaisonInv(liaison)$. Dans notre exemple, $liste\text{-}fournisseurs = \{a_1\}$. Pour chaque élément de la liste, donc ici pour a_1, nous calculons $nbLiens$, le nombre de liaisons de l'architecture d'exécution dont la liaison correspondante est liaison et qui sont liés à a_1 côté fournisseur de service ; dans cet exemple, $nbLiens = 2$. Nous pouvons alors vérifier le respect des cardinalités, qui sont ici satisfaites.

Nous faisons ensuite de même avec les consommateurs de service. Ici, $liste\text{-}consommateurs = \{b_1, b_2\}$ et $nbLiens = 1$ pour b_1 comme pour b_2. Pour ces deux composants, les cardinalités sont également respectées.

Algorithme 5.4 Vérification du respect des cardinalités d'une liaison de l'architecture de conception.

Entrée : $A_c = (C_c, L_c)$, une architecture de conception
Entrée : $A_e = (C_e, L_e)$, une architecture de l'exécution
Entrée : $liaison$
Entrée : $correspondanceLiaisonInv$

1: ▷ vérification côté fournisseur
2: liste-fournisseurs ← {f ∈ C_e | ∃ l ∈ correspondanceLiaisonInv(liaison), (fournisseur(l) = f) }
3: **pour tout** f ∈ liste-fournisseurs **faire**
4: nbLiens ← | { l ∈ correspondanceLiaisonInv(liaison) | fournisseur(l) = f} |
5: **si** nbLiens < liaison.card-min-requis ∨ nbLiens > liaison.card-max-requis **alors**
6: NOTIFIER-ERREUR("cardinalités fournisseur non respectées pour le lien")
7: **fin condition**
8: **fin pour**
9:
10: ▷ vérification côté consommateur
11: liste-consommateurs ← {f ∈ C_e | ∃ l ∈ correspondanceLiaisonInv(liaison), (consommateur(l) = f) }
12: **pour tout** f ∈ liste-consommateurs **faire**
13: nbLiens ← | { l ∈ correspondanceLiaisonInv(liaison) | consommateur(l) = f} |
14: **si** nbLiens < liaison.card-min-fourni ∨ nbLiens > liaison.card-max-fourni **alors**
15: NOTIFIER-ERREUR("cardinalités consommateur non respectées pour le lien")
16: **fin condition**
17: **fin pour**

5.4.5 Synthèse

Cette section vient de décrire les algorithmes qui permettent la vérification de la conformité de l'architecture de l'exécution avec celle de conception. Plus précisément, ceux-ci permettent d'obtenir en plus d'une réponse binaire un ensemble de points localisés où l'architecture de l'exécution est en conflit avec l'architecture de conception, afin de faciliter le processus de mise en conformité.

Pour cela, nous avons proposé une validation en deux étapes :

- la première est effectuée par l'algorithme de correspondance. Il cherche à déterminer pour chaque composant de l'architecture de l'exécution le composant de l'architecture de conception auquel il correspond ;
- dans une seconde étape, la correspondance exacte entre les composants des deux architectures est vérifiée par l'algorithme de vérification. Les erreurs de granularité fine (comme une violation de contrainte sur les propriétés d'un composant ou sur la cardinalité d'un lien) sont alors détectées.

L'algorithme de correspondance effecue une série d'itérations, chacune étant paramétrée à l'aide d'une fonction de correspondance. Celle-ci prend en paramètre une instance de l'architecture de l'exécution et retourne le composant qui lui correspond dans l'architecture de conception. Grâce à ce mécanisme, il est possible de prévoir des fonctions de correspondances spécifiques à des cas d'usage. Dans tous les cas, nous avons proposé trois fonctions de correspondance destinées à couvrir les contextes d'utilisation où la correspondance est décidable.

L'algorithme de validation détecte l'ensemble des violations de contraintes. La liste ainsi établie peut être transmise à un administrateur (par exemple, via Cilia IDE présenté dans le chapitre suivant) ou à un gestionnaire autonomique.

5.5 Conclusion

Ce chapitre nous a permis de décrire les méta-modèles, modèles et algorithmes de notre contribution. Pour cela, nous avons commencé par approfondir la notion de composant en établissant une distinction entre *type composant* et *(objet) composant*, et en introduisant une relation d'instanciation entre les deux. Nous avons vu que les types sont définis en dehors des architectures, contrairement aux composants. Nous avons aussi introduit les notions de propriété, paramètre, variable d'état et contrainte, en mettant l'accent sur leur apports respectifs pour prendre en charge la variabilité ou restreindre celle-ci.

Nous avons commencé par présenter le méta-modèle commun, hérité et rafiné par le méta-modèle de conception, de déploiement et d'exécution. De plus, nous avons donné les principales contraintes OCL qui régissent leur utilisation.

En prenant appui sur cette formalisation, nous avons présenté les algorithmes qui permettent la vérification de la conformité de l'architecture de l'exécution avec celle de conception. Pour cela, nous avons proposé une validation en deux étapes :

- la première est effectuée par l'algorithme de correspondance. Il cherche à déterminer, pour chaque composant de l'architecture de l'exécution, le composant de l'architecture de conception auquel il correspond ;
- dans une seconde étape, la correspondance exacte entre les composants des deux architectures est vérifiée par l'algorithme de vérification. Les erreurs de granularité fine (comme une violation de contraintes sur les propriétés d'un composant ou sur la cardinalité d'un lien) sont alors détectées.

Le chapitre 6 va maintenant présenter l'implantation de notre proposition de thèse et son utilisation dans le cadre de notre validation.

Troisième partie

Implantation et expérimentations

Chapitre 6

Réalisation

Sommaire

6.1 Introduction **158**
6.2 La plate-forme Cilia **160**
6.2.1 Présentation des concepts 160
6.2.2 Exécution 162
6.2.3 Déploiement d'une application Cilia 163
6.3 La base de connaissances **164**
6.3.1 Connaissance de conception 164
6.3.2 Connaissance de l'exécution 164
6.3.3 Algorithme de validation des architectures exécutées 165
6.4 L'atelier Cilia IDE **166**
6.4.1 Introduction 166
6.4.2 Les vues spécifications et implantations de composants 167
6.4.3 La réification des architectures 168
6.4.4 Vues propriétés et erreurs 169
6.5 Le gestionnaire autonomique de déploiement **170**
6.6 Conclusion **172**

6.1 Introduction

L'objectif de cette réalisation est de simplifier les tâches des administrateurs d'applications à base de composants orientés services, en maintenant des liens de traçabilité entre les architectures réalisées à différents moments du cycle de vie de l'application. Pour cela, nous allons utiliser les modèles et algorithmes présentés au chapitre précédent pour :

- définir des architecture de conception ;
- définir des architectures de déploiement ;
- instancer les applications grâce à leur architecture de déploiement ;
- superviser les applications et vérifier leur conformité avec leur architecture de conception ;
- adapter les applications à leur contexte d'exécution, de manière autonomique.

Il nous a donc été nécessaire de nous appuyer sur une plate-forme d'exécution pour laquelle concevoir, déployer, superviser et faire évoluer les architectures. Cette plate-forme devait avoir les caractéristiques suivantes :

- être une plate-forme à composants (paragraphe 2.2.3 page 20), pour offrir un point de vue architectural sur l'application ;
- supporter la dynamicité, c'est-à-dire permettre le déploiement et le retrait des composants lors de l'exécution, ainsi que la modification des liaisons entre ceux-ci. L'état de l'art a montré que cela était possible, notament avec les composants à services (paragraphe 2.3.4 page 33) ;
- être supervisable au niveau composant, et offrir pour chacun d'eux les notions de proprité, de paramètre et de variable d'état.

De plus, pour déclencher les adaptations nécessaires à une modification du contexte d'exécution, la plate-forme doit être en mesure de percevoir celui-ci. Ces travaux étant réalisés dans le cadre du projet FUI MEDICAL centré sur l'informatique pervasive (paragraphe 7.1.2 page 175), la notion de contexte sera pour nous équivalente à l'ensemble des ressources matérielles (capteurs et actionneurs) et logicielles (services *cloud*) disponibles à l'application.

Avec ces contraintes, notre choix de plate-forme s'est porté sur *Cilia Mediation Framework* (aussi appelé plus simplement Cilia) [GG12, Mor13], un *framework open-source* [9], développée au sein de l'équipe Adèle du Laboratoire d'Informatique de Grenoble. L'objectif de Cilia est de simplifier la construction d'applications de médiation. Cilia supporte parfaitement l'évolution à chaud de son architecture à base de composants orientés services dynamiques et est capable de rendre compte de son état interne. De plus, Cilia embarque l'intergiciel RoSe [10], dédié à la découverte et la communication avec des équipements et services distants. Cilia répond donc parfaitement à nos attentes exprimées précédemment.

9. http://wikiadele.imag.fr/index.php/Cilia
10. http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose

L'implantation de notre proposition est résumée sur la figure 6.1. Sur celle-ci, nous retrouvons un ensemble de plates-formes Cilia (section 6.2), chacune disposant du *framework* RoSe afin de percevoir son contexte d'exécution.

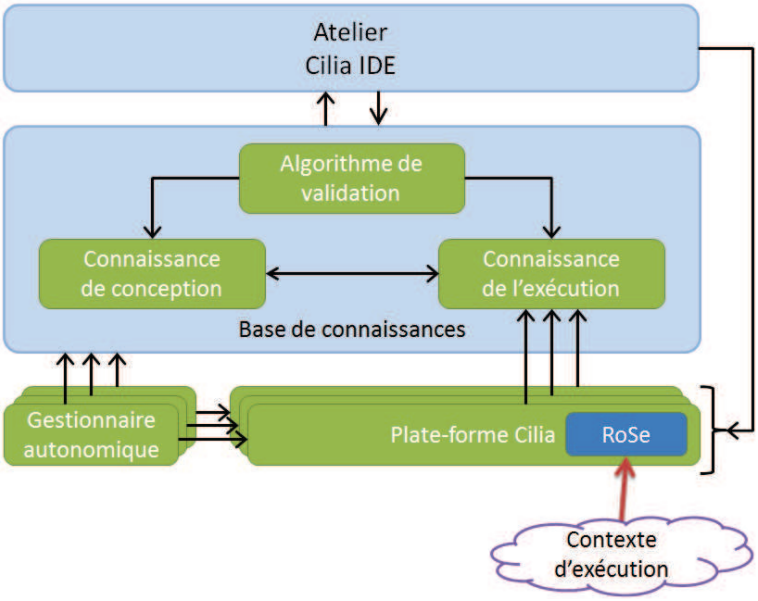

FIGURE 6.1 – Modules de notre réalisation.

La base connaissances (section 6.3 page 164) centralise les architecures de l'exécution des différentes plates-formes Cilia. De plus, elle dispose aussi de la connaissance de conception qui peut être ajoutée soit de manière manuelle, soit à l'aide de Cilia IDE (présenté ci-dessous). La connaissance de conception et celle de l'exécution sont en étroite relation, et l'algorithme de validation permet de vérifier qu'une architecture exécutée est conforme à son architecture de conception.

L'atelier Cilia IDE (section 6.4 page 166) permet de concevoir, de déployer et de superviser des architectures à base de composants Cilia. Il perçoit l'état des plates-formes Cilia via la base de connaissances qu'il peut, par ailleurs, alimenter sur la partie conception. Enfin, Cilia IDE est en mesure d'envoyer des ordres de reconfiguration directement aux plates-formes Cilia.

Les gestionnaires autonomiques (section 6.5 page 170) permettent d'automatiser l'adaptation des plates-formes Cilia aux modifications du contexte d'exécution en déployant le code nécessaire. Contrairement à l'atelier Cilia IDE, les gestionnaires autonomiques ne peuvent que consulter la base de connaissances (il ne peuvent pas la mettre à jour), même si celle-ci est mise à jour indirectement par leur action sur les plates-formes d'exécution.

Nos contributions sont la base de connaissances, l'atelier Cilia IDE et le gestionnaire autonomique (instancié plusieurs fois sur la figure 6.1).

Nous allons maintenant détailler l'ensemble des éléments de la figure 6.1, en commençant par la plate-forme Cilia.

6.2 La plate-forme Cilia

6.2.1 Présentation des concepts

Une application Cilia s'appelle une chaîne de médiation Cilia (ou plus simplement une chaîne). Celle-ci est composée d'un ensemble d'éléments en relation par des liaisons appelées *bindings* (figure 6.2). Nous allons commencer par présenter les deux types d'éléments que l'on peut trouver dans une chaîne (médiateur ou adaptateur) avant de détailler le fonctionnement des *bindings*.

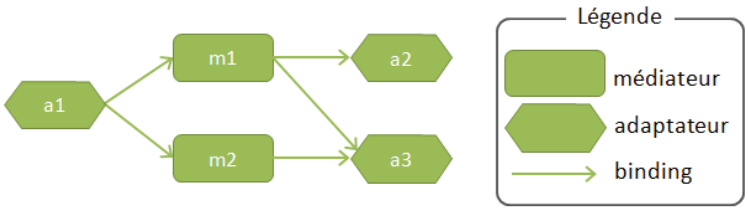

FIGURE 6.2 – Exemple de chaîne de médiation Cilia.

Les médiateurs sont situés au cœur de la chaîne. Chacun d'eux effectue une opération de médiation unitaire, comme une transformation, une agrégation ou une corrélation de données. Un médiateur interagit avec l'extérieur par l'intermédiaire de ports typés qui permettent de recevoir des données (ports d'entrée) ou au contraire d'en diffuser (ports de sortie). Un médiateur peut, par ailleurs, disposer de dépendances de services, ainsi que d'interfaces d'administration.

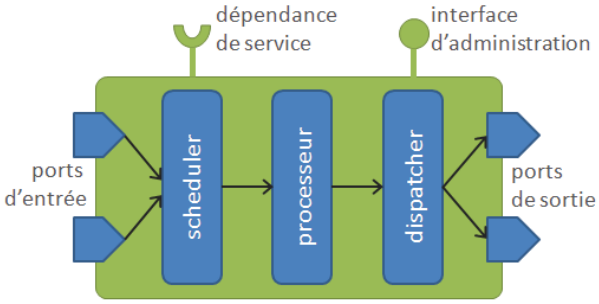

FIGURE 6.3 – Architecture d'un médiateur Cilia.

L'intérieur du médiateur est divisé en trois parties : le *scheduler*, le *processor* et le *dispatcher* (figure 6.3). Le *scheduler* reçoit les données issues des ports du médiateur. Il a pour rôle de décider du moment opportun pour traiter celles-ci. Par exemple, il peut déclencher les traitements à chaque réception de données ou, au contraire, périodiquement. Il peut aussi attendre qu'une donnée ait été reçue sur chacun de ses ports d'entrée. Quand la condition de déclenchement est satisfaite, le *scheduler* envoie les données au *processor*. Celui-ci à pour fonction d'assurer la transformation des données. Le résultat est ensuite envoyé au *dispatcher*

qui décide à la fois du ou des ports de sortie sur lesquels transmettre les données, et du moment opportun pour le faire.

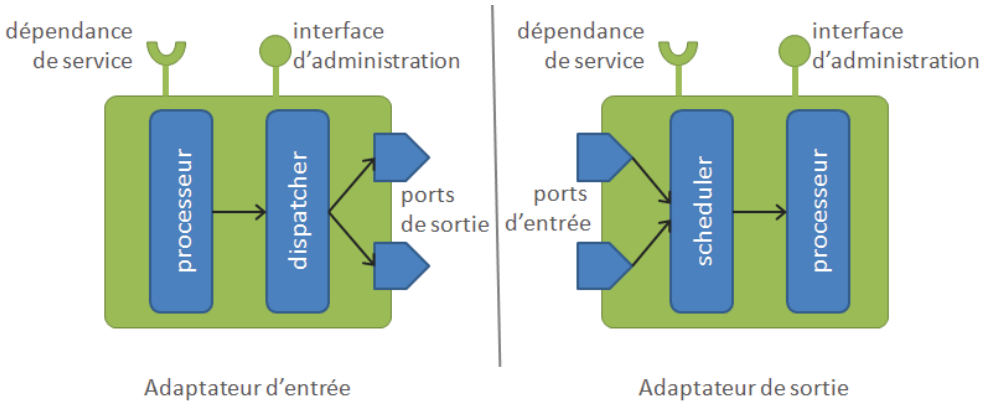

FIGURE 6.4 – Architecture des adaptateurs Cilia.

Les adaptateurs ont pour fonction de prendre en charge la communication avec l'extérieur de la chaîne Cilia. Ces adaptateurs peuvent être de deux types [11] : adaptateurs d'entrée et adaptateurs de sortie (figure 6.4). Les adaptateurs d'entrée permettent de collecter des données à l'extérieur pour les injecter dans la chaîne. Puisqu'ils ne reçoivent pas de données de l'intérieur de la chaîne, ils ne possèdent ni ports d'entrée, ni *scheduler*. Symétriquement, les adaptateurs de sortie transmettent des données à l'extérieur de la chaîne. Ils ne disposent donc ni de *dispatcher*, ni de ports de sortie.

Pour connecter les médiateurs et les adaptateurs, les chaînes Cilia disposent de *bindings* qui relient un port de sortie d'un élément de médiation au port d'entrée d'un autre élément. Ces *bindings* permettent de transmettre des données de manière locale ou distante en spécifiant le protocole de communication utilisé.

Nous venons d'étudier les concepts qui structurent le *framework* de médiation Cilia, et qui permettent ainsi de définir des chaînes de médiation. Nous allons maintenant voir comment sont exécutées celles-ci.

11. Nous ne parlerons pas ici des adaptateurs d'entrée-sortie, plus complexes et en cours de spécification.

6.2.2 Exécution

L'architecture de la plate-forme d'exécution Cilia est présentée à la figure 6.5. Au plus bas niveau, l'exécution repose sur la machine virtuelle Java™. Sur la base de celle-ci, OSGi™ apporte le chargement et le déchargement dynamique de code, et Apache Felix iPOJO ajoute les composants orientés services. La plate forme Cilia s'appuie ensuite à la fois sur Apache Felix iPOJO et sur RoSe, qui fournit des facilités de communication avec des entités distantes.

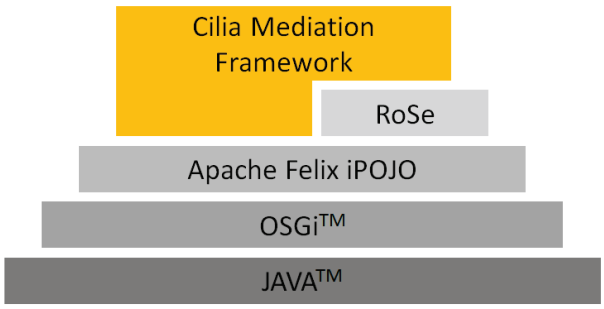

FIGURE 6.5 – Architecture de *Cilia Mediation Framework*.

A l'exécution, Cilia configure et exécute un ensemble de composants iPOJO. Cela offre la possibilité d'adjoindre à ceux-ci un ensemble de *handlers* pour ajouter des capacités de supervision sous la forme de variables d'état [Mor13]. Cependant, cela entraîne aussi la multiplication des instances iPOJO. Par exemple, un médiateur est exécuté sous la forme d'au moins cinq instances iPOJO : trois pour le *scheduler*, le *processor* et le *dispatcher* et deux autres pour un port d'entrée et de sortie. Manipuler une chaîne de médiation à ce niveau d'abstraction peut donc se révéler complexe et risqué. C'est la raison pour laquelle Cilia propose une API réflexive [Mae87] de plus haut niveau, qui s'appuie sur la séparation entre un *base-level* et un *meta-level* (figure 6.6) [GGMD+11].

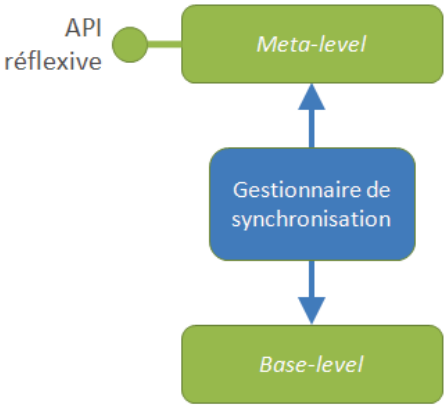

FIGURE 6.6 – Architecture réflexive de *Cilia Mediation Framework* (modèle simplifié).

Pour Cilia, les instances iPOJO sont situées au *base-level*. Parallèlement, le *meta-level* dispose d'une description de ce qui est exécuté en termes de chaînes, de médiateurs et d'adap-

tateurs. Pour reprendre notre exemple, les cinq composants du médiateur se trouvent ainsi abstraits dans un unique objet. Un gestionnaire de synchronisation a pour rôle de conserver le *base-level* et le *meta-level* en cohérence. Pour cela, il effectue une réification du *base-level* dans le *meta-level*. En sens inverse, toute modification sur le *meta-level* est propagée par ce gestionnaire au *base-level*, sous la forme d'ordres de création, de modification et de mises à jour au niveau des composants iPOJO.

6.2.3 Déploiement d'une application Cilia

Le déploiement d'une application Cilia nécessite deux types d'artéfacts :

- le code source des composants de médiation (médiateurs et adaptateurs) ;
- un fichier XML (dscilia) qui décrit la configuration des composants et leur assemblage.

Le code source est du code Java™, qui est compilé et packagé dans un ensemble de JAR. Cela permet de partager facilement les implantations de composants de médiation entre plusieurs projets. Le fichier XML, qui porte l'extension dscilia est, quant à lui, composé de deux parties. Une première décrit la configuration initiale des instances de composants qui devront être créées. Une seconde décrit les *bindings* qui permettent de relier ces instances au niveau de leurs ports.

Sur la plate-forme Cilia, un gestionnaire écoute les événements de création ou de modification de fichiers. Quand un fichier XML est créé, ce gestionnaire regarde s'il s'agit d'une description de chaîne Cilia. Dans ce cas, il utilise l'API réflexive afin d'instancier et de configurer cette nouvelle chaîne. En cas de simple mise à jour du fichier, il effectue une mise à jour de la chaîne de la même manière. Notons enfin qu'il est aussi possible de créer une chaîne directement à l'aide de l'API réflexive.

6.3 La base de connaissances

Nous avons vu dans l'introduction que la base de connaissances est centrale dans notre approche. En effet, elle est absolument nécessaire à l'atelier Cilia IDE et aux gestionnaires autonomiques. Cette connaissance peut être relative à la conception, à l'exécution ou à la validation des architectures exécutées.

6.3.1 Connaissance de conception

La connaissance de conception est composée :

- des spécifications et des implantations de composants ; c'est-à-dire des *TypesComposants* de la proposition ;
- des architectures de conception et de déploiement.

Pour réifier cette connaissance, les différents artéfacts sont introspectés. En particulier, les archives JAR des implantations de composants Cilia sont décompressées en mémoire afin de découvrir leur contenu. Cela permet de connaître leur structure interne avec précision. Les architectures, quant à elles, sont exprimées sous forme de fichiers XML. Leur réification est donc plus aisée.

6.3.2 Connaissance de l'exécution

Nous avons vu en introduction que la connaissance de l'exécution est double :

- la connaissance interne de la plate-forme d'exécution ;
- le contexte d'exécution, une connaissance externe à la plate-forme d'exécution.

Connaissance de la plate-forme d'exécution

Cette connaissance de l'exécution est obtenue auprès de la plate-forme Cilia via l'API réflexive qui permet d'avoir accès à l'état du *meta-level*. Cette API est accessible soit directement par l'invocation de méthodes sur la JVM, soit par l'intermédiaire d'une interface REST qui permet aussi la communication depuis des hôtes distants.

Dans notre cas, nous avons utilisé l'interface REST. Grâce à celle-ci, il nous est possible de connaître très simplement la composition et l'état des chaînes de médiation Cilia. On peut notamment obtenir l'ensemble des propriétés de celle-ci : état des médiateurs, des adaptateurs, activation et lecture des variables d'état, etc.

A titre d'exemple, le Code source 6.1 donne le résultat sous forme JSON d'une requête demandant la composition d'une petite chaîne de médiation que nous avons utilisée durant nos tests. Par souci de lisibilité, des retours à la ligne et des espaces ont été ajoutés.

```
1  {
2    "ID":"test-rest-api",
3    "Mediators":["filter","ts-enricher","transformer","loc-enricher","
       enricher"],
4    "Adapters":{
5      "in-out":[],
6      "out-only":["event-webservice"],
7      "in-only":["presence-collector"]
8    },
9    "Bindings":[
10     {"to":"loc-enricher:in","from":"presence-collector:out"},
11     {"to":"enricher:in","from":"filter:out"},
12     {"to":"transformer:in","from":"ts-enricher:out"},
13     {"to":"filter:in","from":"loc-enricher:out"},
14     {"to":"ts-enricher:in","from":"enricher:out"},
15     {"to":"event-webservice:in","from":"transformer:out"}
16   ]
17 }
```

Code source 6.1 – Exemple de code JSON retourné par une requête REST.

Connaissance du contexte d'exécution

La connaissance du contexte est un élément clé dans notre approche. En effet, que ce soit à l'aide d'un atelier ou avec un gestionnaire autonomique, notre objectif est d'adapter un système à son environnement dans le respect de ses contraintes de conception.

Dans notre cas, le contexte sera un ensemble de ressources externes. Grâce au *middleware* RoSe, la plate-forme gère l'accès aux équipements et met à disposition du programmeur des *proxies* de haut niveau. Leur cycle de vie suit celui des équipements : un *proxy* est créé quand un équipement est détecté et il est détruit quand l'équipement disparaît. Pour l'application, il s'agit du seul point d'accès aux équipements. La connaissance du contexte se limite donc pour nous à la liste des *proxy* RoSe disponibles. Notons que cette notion de contexte, non centrale dans le cadre de cette thèse, mériterait d'être étudiée de manière approfondie.

6.3.3 Algorithme de validation des architectures exécutées

La dernière pièce de la base de connaissances est l'algorithme qui permet de valider une architecture de l'exécution au regard de son architecture de conception. Cet algorithme a été implanté en Java™. Il est important de noter que son implantation a été rendue possible grâce aux liens qui rapprochent la connaissance de conception de celle d'exécution. En cela, le respect du méta-modèle présenté dans la partie proposition a été une condition de possibilité de l'implantation de cet algorithme.

Du point de vue du code, l'implantation ne représente que 423 lignes de code (non vides et hors commentaires) réparties dans les classes *RuntimeToRefArchManager*, *RuntimeToRefArchCommon* et *RuntimeToRefArchChecker*. En effet, une partie du travail des algorithmes est

proposée directement au niveau du modèle afin de faciliter sa manipulation, que ce soit par l'algorithme, l'atelier ou les gestionnaires autonomiques.

Maintenant que nous avons présenté le socle commun de la base de connaissances, voyons la manière avec laquelle celui-ci est utilisé, en commençant par l'atelier Cilia IDE.

6.4 L'atelier Cilia IDE

6.4.1 Introduction

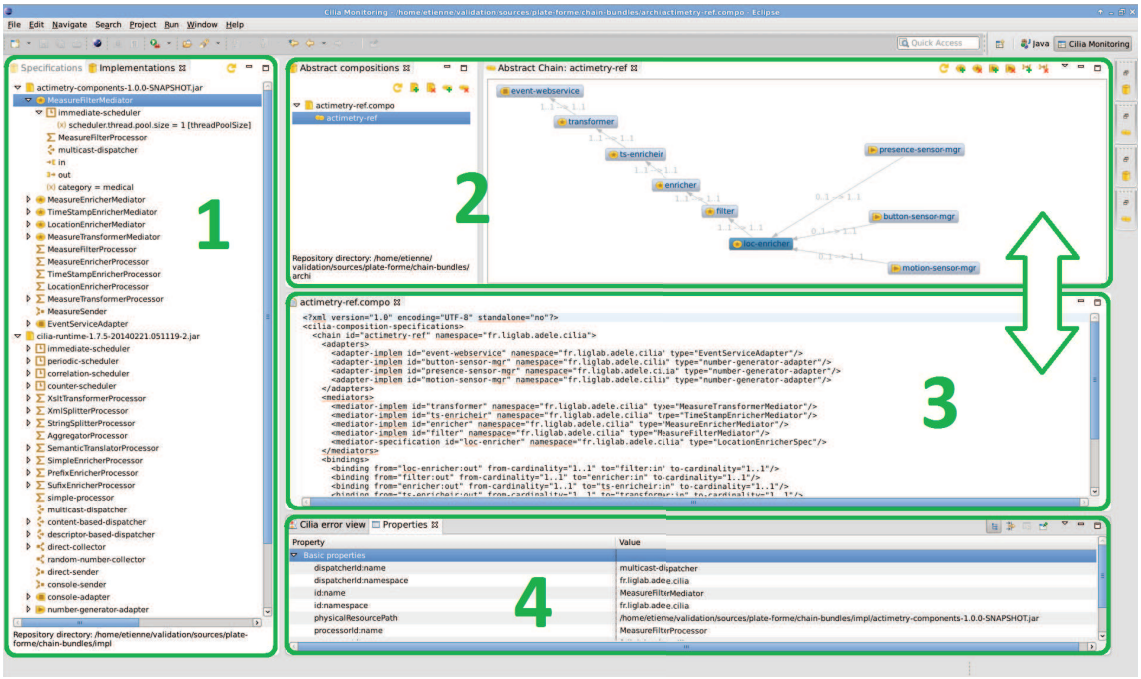

FIGURE 6.7 – L'atelier Cilia IDE.

Cilia IDE est un atelier basé sur Eclipse qui permet de développer, de déployer et de superviser des chaînes de médiation Cilia. Il dispose de nombreuses vues qui ne peuvent pas être toutes affichées simultanément si on désire conserver une lisibilité optimale. Cependant, l'atelier peut dans tous les cas être divisé en quatre zones (figure 6.7) :

- **zone 1 :** affichage des réifications des spécifications et des implantations de composants (deux vues) qui seront référencés par la suite dans les architectures. Cela correspond aux *TypeComposant* de notre proposition ;
- **zone 2 :** affichage des réifications d'une ou plusieurs architectures, à base des *Objet-Composant* de la proposition ;
- **zone 3 :** code source des architectures en cours d'édition ;
- **zone 4 :** propriétés de l'élément sélectionné et erreurs diagnostiquées (deux vues).

Notons que les zones 2 et 3 sont synchronisées : toute modification du code se traduit par une mise à jour de la réification de celui-ci et toute modification graphique de l'architecture est traduite en retour dans le code. Cette synchronisation bidirectionnelle est prévue pour conserver le code qui ne serait pas traduit dans la réification (des extensions de Cilia peuvent ajouter des balises aux fichiers XML).

Nous allons maintenant présenter plus en détails le fonctionnement général des zones 1, 2 et 4, la zone 3 étant un éditeur fourni par Eclipse.

6.4.2 Les vues spécifications et implantations de composants

La zone 1 de la figure 6.7 affiche une réification des implantations et des spécifications de composants, c'est-à-dire des *TypesComposants* de la proposition. Pour cela, Cilia IDE s'appuie sur la connaissance de conception décrite dans la partie précédente. Ainsi, ces vues permettent, non seulement, de lister les composants disponibles, mais aussi de connaître leur structure interne avec précision. Sur le détail de la figure 6.8, nous pouvons par exemple constater que le fichier *actimetry-components-1.0.0-SNAPSHOT.jar* définit, entre autres, l'implantation du médiateur *MessureFilterMediator*, qui est composée d'un *immediate-scheduler* (dont on peut configurer la taille du *pool* de *threads* grâce à un paramètre), d'un *MeasureFilterProcessor* et d'un *MulticastDispatcher*. Cette implantation a aussi deux interfaces (un port d'entrée nommé *in*, un port de sortie nommé *out*) et une propriété *category* qui porte la valeur *medical*.

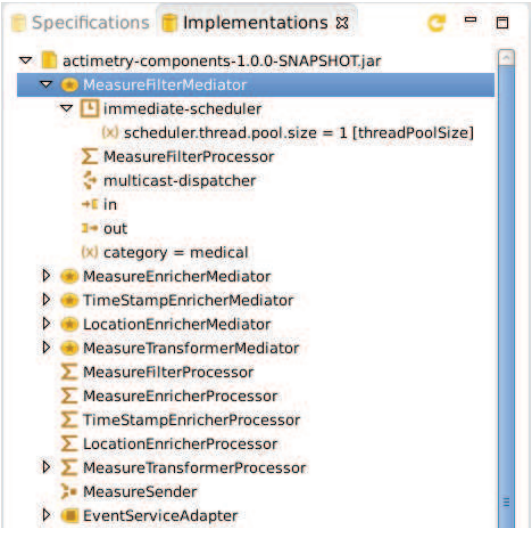

FIGURE 6.8 – Présentation de la vue *Implementations* de Cilia IDE.

6.4.3 La réification des architectures

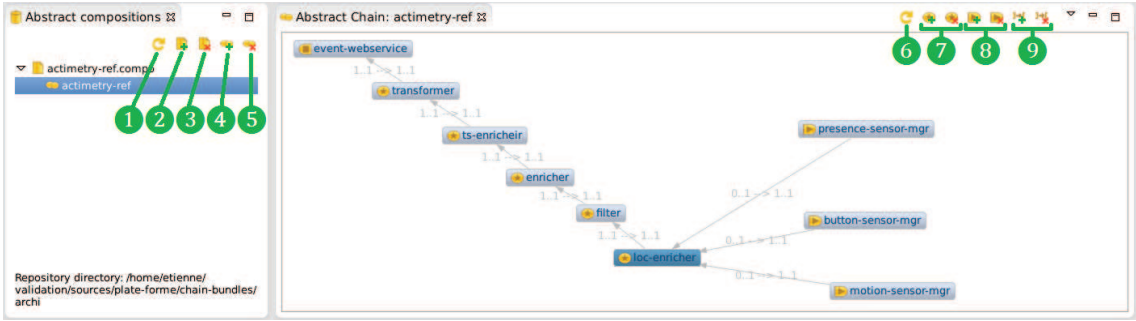

FIGURE 6.9 – Présentation d'une vue architecture de Cilia IDE.

Pour afficher de manière graphique des architectures, Cilia IDE propose un couple de vues par type d'architecture. La figure 6.9 montre les deux vues qui permettent d'afficher une architecture de conception. Sur la gauche, un diagramme en forme d'arbre liste les architectures disponibles. En haut de la vue, une série de boutons permettent de rafraîchir la vue ❶, de créer ❷ et de détruire ❸ un fichier contenant des architectures, ou de créer ❹ et détruire ❺ une architecture au sein d'un fichier. Quand on sélectionne une architecture dans cette vue, celle-ci est affichée sur la vue de droite, qui permet la visualisation et la modification de celle-ci. Cette seconde vue dispose elle-aussi d'un ensemble de boutons d'action qui permettent de rafraîchir la vue ❻, d'ajouter ou de supprimer un médiateur ❼, d'ajouter ou de supprimer un adaptateur ❽, d'ajouter ou de supprimer un *binding* ❾. Notons que les médiateurs et les adaptateurs dont il est ici question correspondent aux *Composants* définis dans notre proposition.

Cette réification reste parfaitement synchronisée avec le code source qui peut être édité dans la zone 3 de la figure 6.7 page 166. Pour maintenir cette cohérence, chaque action effectuée graphiquement produit en fait une modification au niveau du code suivi d'une nouvelle réification de l'architecture qui provoque une mise à jour de l'affichage de celle-ci. De même, chaque édition entraîne une mise à jour de la réification et donc de son affichage.

6.4.4 Vues propriétés et erreurs

La zone 4 de Cilia IDE (figure 6.10) permet d'afficher la vue *Properties* (Propriétés) ou la vue *Cilia error view* (Erreurs Cilia).

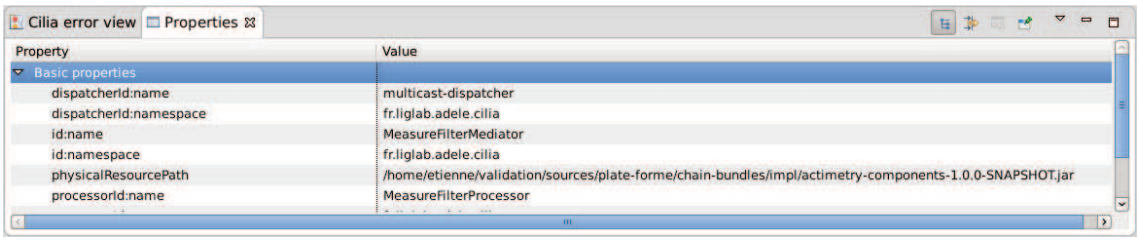

FIGURE 6.10 – Propriétés de l'implantation *MeasureFilterMediator*.

Quand on clique sur un élément de l'IDE (élément d'un arbre, élément d'un graphe), la liste des propriétés qui lui sont associées dans la base de connaissances est automatiquement affichée dans la vue *Properties*. Par exemple, la sélection de *MeasureFilterMediator* dans la vue *Implementations* (figure 6.8 page 167) entraîne l'affichage des propriétés de cet élément (figure 6.10).

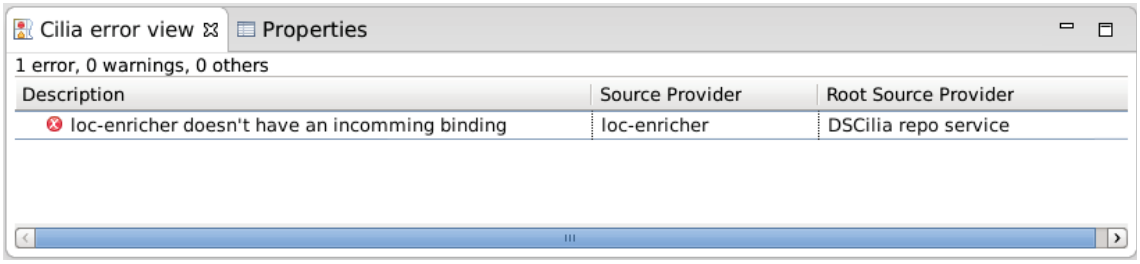

FIGURE 6.11 – Vue *Cilia error view*.

La vue *Cilia error view* (figure 6.11) permet, quant à elle, de centraliser les erreurs détectées dans la base de connaissances. En particulier, elle affiche les erreurs diagnostiquées lors de la validation de l'architecture exécutée selon l'algorithme présenté à la fin de notre proposition.

6.5 Le gestionnaire autonomique de déploiement

Le rôle du gestionnaire autonomique est de compléter la chaîne de médiation en lui ajoutant des morceaux en fonction des nouvelles ressources disponibles sur la plate-forme d'exécution. Pour structurer les étapes nécessaires à cette mise à jour, nous avons suivi l'architecture MAPE-K [IBM06] proposée par IBM (paragraphe 3.4.3 page 76) :

- **surveillance (M) :** notre gestionnaire autonomique va écouter les modifications apportées à la connaissance de l'environnement d'exécution. Quand un dispositif éligible pour l'application est réifié par RoSe et enregistré dans la base de connaissances **(K)**, la tâche de surveillance va appeler celle d'analyse ;
- **analyse (A) :** si le dispositif peut être pris en charge par l'application, on regarde si le code nécessaire est déjà déployé. Pour cela, le gestionnaire autonomique dispose d'une base de connaissances **(K)** propre que nous présenterons par la suite. Si le code nécessaire est absent, la tâche de planification est lancée ;
- **planification (P) :** grâce à sa base de connaissances **(K)**, le gestionnaire autonomique va trouver l'adresse à laquelle télécharger le code qu'il faudra installer. Il ne reste plus alors qu'à lancer l'exécution ;
- **exécution (E) :** cette dernière tâche assure trois missions. Tout d'abord, le code est téléchargé depuis l'Internet, ensuite, il est intégré à la plate-forme Cilia qui supporte nativement le déploiement à chaud [GG12], enfin, la nouvelle architecture exécutée est validée par rapport à l'architecture de conception présente dans la base de connaissances **(K)**.

Concernant la logique de fonctionnement de ce gestionnaire autonomique, nous avons fait le choix de ne jamais retirer de code déployé, même si celui-ci n'est plus nécessaire. En effet, cela permet d'éviter des installations/désinstallations fréquentes en cas de disponibilité fluctuante d'un équipement (comme un *smartphone*, qui entre et sort de l'habitat) ou en cas de changement de piles des dispositifs. Notons qu'un gestionnaire autonomique de désinstallation pourrait être implanté afin de retirer le code quand celui-ci n'est plus nécessaire durant une longue durée.

Nous pouvons constater que la base de connaissances est sollicitée tout au long des tâches MAPE. Nous avons déjà décrit la manière avec laquelle l'architecture de conception est formalisée dans notre approche, ainsi que le fonctionnement du processus de validation de l'architecture exécutée. Nous allons donc nous intéresser maintenant à la connaissance nécessaire à l'analyse et à la planification.

Cette connaissance est décrite dans un ensemble de fichiers XML qui sont pris en compte par la plate-forme d'exécution sitôt qu'ils sont déposés sur celle-ci. Chaque fichier contient une ou plusieurs entrées qui permettent d'associer une URL de téléchargement à une ou plusieurs interfaces de services qui correspondent aux équipements réifiés par RoSe.

Le Code source 6.2 est le fichier de configuration du gestionnaire autonomique. Sur celui-ci nous avons l'adresse de téléchargement du code source à déployer pour les trois

```
<db>
  <id name="push-button-fragment" url="http://medical.imag.fr/documents/
      am/push.button.adapter.dp-1.2.6-SNAPSHOT.dp">
    <interface>fr.liglab.adele.icasa.device.button.PushButton</interface>
  </id>
  <id name="presence-sensor-fragment" url="http://medical.imag.fr/
      documents/am/presence.sensor.adapter.dp-1.2.6-SNAPSHOT.dp">
    <interface>fr.liglab.adele.icasa.device.sensor.presence.
        PresenceSensor</interface>
  </id>
  <id name="motion-sensor-fragment" url="http://medical.imag.fr/documents
      /am/motion.sensor.adapter.dp-1.2.6-SNAPSHOT.dp">
    <interface>fr.liglab.adele.icasa.device.sensor.motion.MotionSensor</
        interface>
  </id>
</db>
```

Code source 6.2 – Fichier de configuration du gestionnaire autonomique

équipements qui seront pris en compte dans notre validation. Ce système permet donc de paramétrer facilement notre gestionnaire autonomique en lui fournissant le supplément de connaissances dont il a besoin.

6.6 Conclusion

La réalisation conduite dans le cadre de cette thèse s'est appuyée sur le *framework* de médiation Cilia. Elle a ajouté à celui-ci une base de connaissances clairement formalisée, utilisée par un atelier (Cilia IDE) ainsi que des gestionnaires autonomiques.

	Base de connaissances, modèles, parseurs, bus de notification	**Cilia IDE**	**Gestionnaire autonomique**	**TOTAL**
Nombre de classes Java™	137	121	4	**262**
Nombre de lignes de code	10939	8379	217	**19535**

Tableau 6.1 – Nombre de classes et de lignes de code de notre réalisation .

Le nombre de lignes de code écrites (non vides) ainsi que le nombre de classes correspondantes est donné dans le tableau 6.1. Il peut sembler surprenant que la base de connaissances compte plus de 10000 lignes de code. Cela s'explique en raison de la présence non seulement de modèles, mais aussi et surtout de tout un ensemble de mécanismes pour en assurer l'exploitation : *parseurs* pour introspecter les fichiers JAR, bus de notification, algorithmes de fusion de modèles pour calculer des différences lors de mises à jour, etc.

A contrario, l'atelier Cilia IDE surprend, pour sa part, par son relativement faible nombre de lignes de code, au vu des nombreuses fonctionnalités implantées : plusieurs dizaines de boîtes de dialogue, une dizaine de vues,... En fait, cet IDE est un ensemble de *plugins* ajoutés à l'IDE Eclipse. Nous avons donc bénéficié d'un ensemble très important de mécanismes qui nous ont permis d'écrire relativement peu de code par rapport au résultat obtenu.

Enfin, le gestionnaire autonomique est assez petit, d'autant plus qu'une partie du code sert à la gestion du chargement et du déchargement dynamique des fichiers XML sur la plate-forme.

Chapitre 7

Validation

Sommaire

7.1 Introduction **174**
7.1.1 Le domaine du *Smart Home* 174
7.1.2 Le projet FUI MEDICAL 175
7.1.3 Présentation de la validation 176
7.2 L'application Actimétrie **177**
7.2.1 Objectif et fonctionnement du service 177
7.2.2 Architecture du service sur la *home gateway* 180
7.2.3 Déploiement et mise à jour du service 182
7.2.4 Synthèse 183
7.3 Développement et supervision du service Actimétrie avec Cilia IDE . **184**
7.3.1 Architecture de conception 184
7.3.2 Architecture de déploiement 186
7.3.3 Déploiement et supervision du service 188
7.3.4 Synthèse 192
7.4 Administration autonomique **193**
7.5 Tests de performance **194**
7.5.1 Utilisation des liaisons avec une architecture de type chaîne . . 194
7.5.2 Utilisation des types 196
7.5.3 Interprétation des résultats obtenus 197
7.6 Conclusion **197**

7.1 Introduction

7.1.1 Le domaine du *Smart Home*

L'objectif du *Smart Home* et de fournir des services dans un habitat à l'aide d'un ensemble de capteurs et d'actionneurs. Ces services peuvent concerner des domaines extrêmement divers, tels que la sécurité des biens et des personnes, le suivi et l'optimisation de la consommation énergétique des équipements, le confort avec, par exemple, la régulation du chauffage ou de l'éclairage, le divertissement avec les équipements multimédia ou encore la santé. Ces services à l'habitat envisagés aujourd'hui dans un contexte très connecté s'inscrivent dans le prolongement de la domotique traditionnelle en tentant de pallier ses limites.

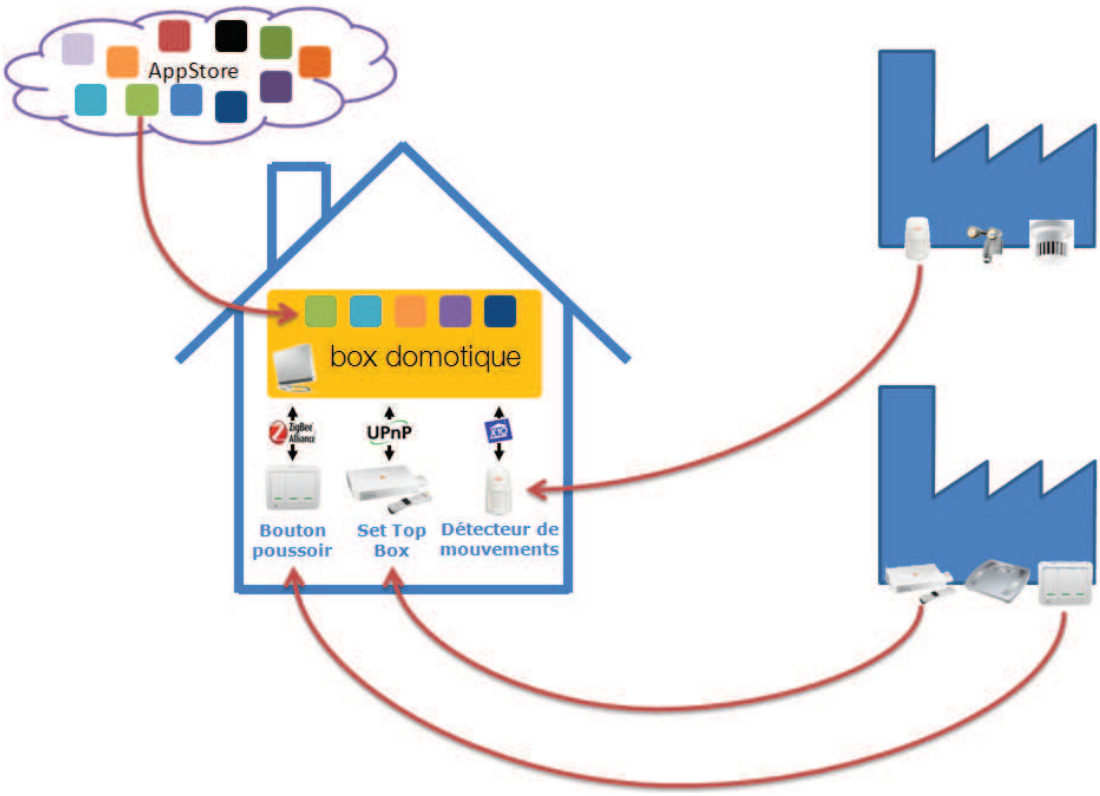

FIGURE 7.1 – Un défi du *Smart Home*.

La domotique s'est développée depuis de nombreuses années en s'appuyant sur l'électronique analogique puis numérique. L'exemple le plus emblématique est celui des centrales d'alarmes, extrêmement répandues. Dans ce cas, il n'y a pas de séparation claire entre l'application et les équipements qui communiquent dans un environnement cloisonné. Si cela offre de la simplicité et de la robustesse, la redondance des équipements est inévitable dès que l'on installe plusieurs services à l'habitat, ce qui entraîne un surcoût global et la supervision isolée de chaque application.

Plus récemment, des équipementiers ont proposé des *home gateway* (MyHOME domotique de Legrand, Blyssbox de Castorama,...), qui permettent à plusieurs applications de se partager des dispositifs communicants. Cela permet de répondre aux limitations de l'approche précédente, en offrant un partage d'équipements et une supervision unifiée de l'ensemble des applications dans un cadre contrôlé par le fournisseur.

Mais cette seconde approche n'est aujourd'hui plus suffisante. En effet, nous assistons à l'entrée dans l'habitat d'un nombre de plus en plus important d'objets communicants : montres connectées, pèse-personne, tensiomètres,... Ces dispositifs proviennent d'équipementiers différents, proposent des protocoles de communication hétérogènes et une manière de représenter les données parfois incompatible (figure 7.1). L'approche précédente, centrée sur la connaissance et la maîtrise des équipements d'un unique fournisseur n'est donc plus adaptée à ce nouvel écosystème d'équipements. De plus, les usagers trouvent maintenant normal de pouvoir choisir leurs applications dans le catalogue d'un *app store*, dans lequel celles-ci peuvent être publiées par différents éditeurs.

7.1.2 Le projet FUI MEDICAL

L'objectif du projet FUI MEDICAL [12] est de concevoir un intergiciel pour le *Smart Home* qui facilite le développement et l'administration d'applications intégrant des objets communicants hétérogènes (fournisseurs, protocoles de communication, schémas de données). C'est un projet collaboratif FUI (Fonds Unique Interministériel) labélisé Minalogic, regroupant Orange Labs (porteur du projet), l'université Joseph Fourrier, Télécom ParisTech et la société ScalAgent D.T. Avec ses partenaires, le projet FUI MEDICAL a livré la plate-forme iCASA [13] [ECL14] en se focalisant sur deux domaines :

- le développement et l'exécution d'applications pervasives à base d'objets communicants, par Orange Labs et l'université Joseph Fourrier ;
- la problématique de distribution, grâce à OW2 Joram [14] et à Cube [Deb14], qui sont les contributions respectives de ScalAgent D.T. et Télécom ParisTech.

La plate-forme logicielle iCASA peut-être embarquée dans la *home gateway* qui accueille les applications, ou être déployée entre la *home gateway* et des serveurs distants (*cloud*). Ces applications offrent des services à l'habitat, en partageant des ressources internes à la *home gateway* (mémoire, processeur, stockage...) et externes (capteurs, actionneurs, services cloud,...). La plate-forme iCASA est architecturée en couches, respectant ainsi le principe de séparation des préoccupations.

Au niveau le plus bas, une machine virtuelle Java™ (JVM) permet l'exécution de la pile logicielle et offre le support de la concurrence (figure 7.2). En s'appuyant sur celle-ci, OSGi™ et Apache Felix iPOJO apportent la capacité de chargement et de déchargement dynamique de code, ainsi que la notion de composants orientés services dynamiques, que nous avons

12. http://medical.imag.fr
13. https://adeleresearchgroup.github.io/iCasa/
14. http://joram.ow2.org/

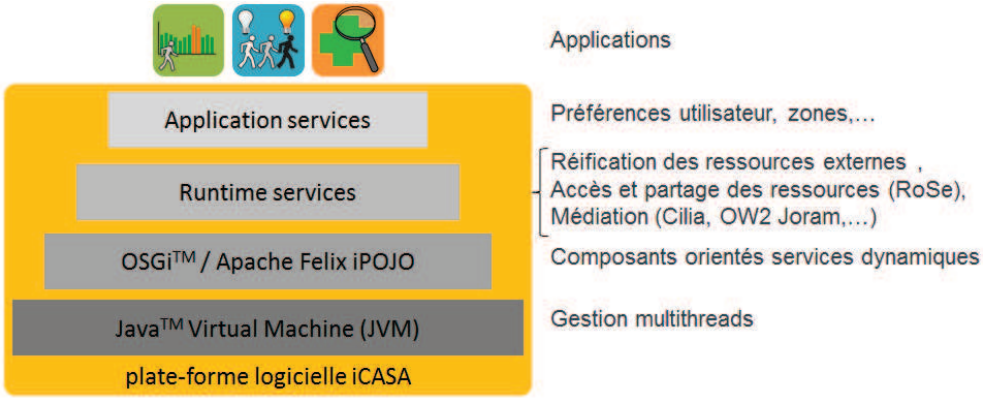

FIGURE 7.2 – Architecture de la plate-forme iCASA.

présentés dans l'état de l'art. Ce socle permet de bâtir un ensemble de services techniques (*Runtime Services*) qui prennent en charge la découverte, l'accès et la réification des ressources externes (avec le *middleware* RoSe) et la gestion des accès, ainsi qu'un *middleware* de médiation. Enfin, les *Application Services* sont un ensemble de services directement accessibles par les applications qui permettent, par exemple, de définir des zones dans l'habitat ou de gérer des préférences utilisateur.

A l'aide de cet ensemble de *middleware*, la plate-forme iCASA offre à l'utilisateur un panorama complet et unifié de son écosystème numérique. Elle lui permet de surveiller l'état des tous ses équipements et de donner aux applications l'accès à ceux-ci. Côté développeur, la plate-forme iCASA facilite l'effort de développement des applications s'intégrant avec des ressources externes, en l'occurrence des dispositifs électroniques. Elle apporte au développeur d'applications une interface de haut niveau qui lui permet d'interagir avec les équipements sans se soucier de leur technologie et de leur protocole. De plus, elle offre des mécanismes de notifications, qui permettent aux applications d'être prévenues dans le cas où un équipement devient disponible ou, au contraire, disparaît.

7.1.3 Présentation de la validation

Avec les mécanismes que nous venons de présenter, la plate-forme iCASA permet aux applications pervasives d'être notifiées de l'arrivée ou du départ d'équipements grâce au *middleware* RoSe. En raison de la forte hétérogénéité fonctionnelle des équipements, cette modification de disponibilité de ressources peut nécessiter l'ajout d'un morceau d'application qui est fonction de l'équipement considéré. Les applications peuvent donc évoluer, en modifiant jusqu'à leur structure, afin de prendre en compte le nouvel écosystème d'équipements.

Dans le cadre du *Smart Home*, il est nécessaire que ces évolutions soient mises en œuvre sans intervention de l'utilisateur. En effet, celui-ci n'est pas expert des applications qui sont exécutées chez lui et il est, par ailleurs, non réaliste de devoir reconfigurer manuellement une application à chaque départ ou arrivée d'équipement. Pour automatiser ce processus

d'adaptation, il est possible d'implanter un gestionnaire selon les concepts de l'informatique autonomique (chapitre 3 page 59). Celui-ci aura la charge d'adapter de manière réactive et transparente l'application aux modifications de disponibilités des ressources en adaptant jusqu'à l'architecture de l'application.

Pour démontrer la validité de notre approche, nous nous appuierons sur l'application « Actimétrie » [ZFA14] (section 7.2) du projet FUI MEDICAL. Celle-ci est construite à l'aide du *framework* de médiation à composants dynamiques Cilia, qui permet de gérer l'ajout et la suppression de code à l'exécution et de modifier ainsi son architecture. Notre validation aura deux volets complémentaires :

- **un atelier de développement, déploiement et supervision** (section 7.3 page 184) des applications de médiation Cilia qui permet à un administrateur d'investiguer finement les relations entre l'architecture de conception et l'architecture exécutée, tout en prenant en compte les relations entre les différents types de composants. Cet atelier permettra de valider les concepts de notre approche.
- **un gestionnaire autonomique** (**??** page ??), capable de modifier l'application pour l'adapter lors de son exécution et valider les modifications apportées à celles-ci. Celui-ci répondra au besoin d'automatisation de l'administration de l'application Actimétrie.

Mais avant de présenter ces deux parties de notre validation et de conclure par des métriques de performance, nous allons commencer par détailler l'objectif et le fonctionnement de ce service, son architecture et sa mise à jour.

7.2 L'application Actimétrie

7.2.1 Objectif et fonctionnement du service

Certaines maladies invalidantes se développent lentement et sont parfois difficilement détectables pour le patient lui-même, pour son entourage et même pour un médecin. Or, les traitements de nombreuses maladies sont d'autant plus efficaces qu'ils sont prescrits précocement. En permettant une détection au plus tôt de ces maladies, le service Actimétrie a pour ambition d'améliorer la prise en charge de personnes isolées ou fragilisées, et de favoriser ainsi leur maintien à domicile sans sacrifier le suivi et la santé des patients.

Pour effectuer sa détection de signes avant-coureurs, le service Actimétrie part du constat suivant : en temps normal, les personnes âgées ou fragilisées ont des vies extrêmement ritualisées avec des habitudes solidement ancrées, que ce soit au niveau du temps de sommeil, de l'horaire des repas, de la structuration des activités au cours d'une journée. En cas de début de maladie, ces habitudes se trouvent lentement modifiées dans leur globalité et présentent des variations importantes à certains moments. Par exemple, un patient pourra se lever plusieurs fois par nuit pour aller manger ou décaler de manière progressive son horaire de coucher. Le service Actimétrie va s'appuyer sur ce constat pour détecter au plus tôt des anomalies à partir de modification des habitudes du patient.

Pour cela, le service va dans un premier temps apprendre les habitudes d'une personne. Par la suite, il sera capable de comparer ses activités avec cette référence. Dans le cas où une modification du rythme de vie de la personne est détectée, une alerte peut être levée auprès du médecin traitant ou de l'entourage, afin qu'ils puissent avoir une attention particulière. La faisabilité et l'intérêt d'un tel service ont été démontrés dans le cadre de l'expérimentation du système MAPA [NQB+09, NQB+10, NBT+11] (Mesure de l'Activité d'une Personne seule A domicile) en partenariat avec Mondial Assistance [15].

La perception des habitudes de la personne est effectuée au travers de son interaction avec l'écosystème numérique de son habitat : utilisation de la *set top box*, manipulation d'interrupteurs connectés, passage devant des détecteurs de présence ou activation des détecteurs d'ouverture de porte. Les événements générés par ces équipements sont agrégés et permettent à l'application de localiser la personne dans son habitat et de déduire des informations sur son activité courante. Par exemple, quand elle change de chaîne sur la télévision, il est réaliste de faire l'hypothèse qu'elle se trouve devant cette même télévision à cet instant.

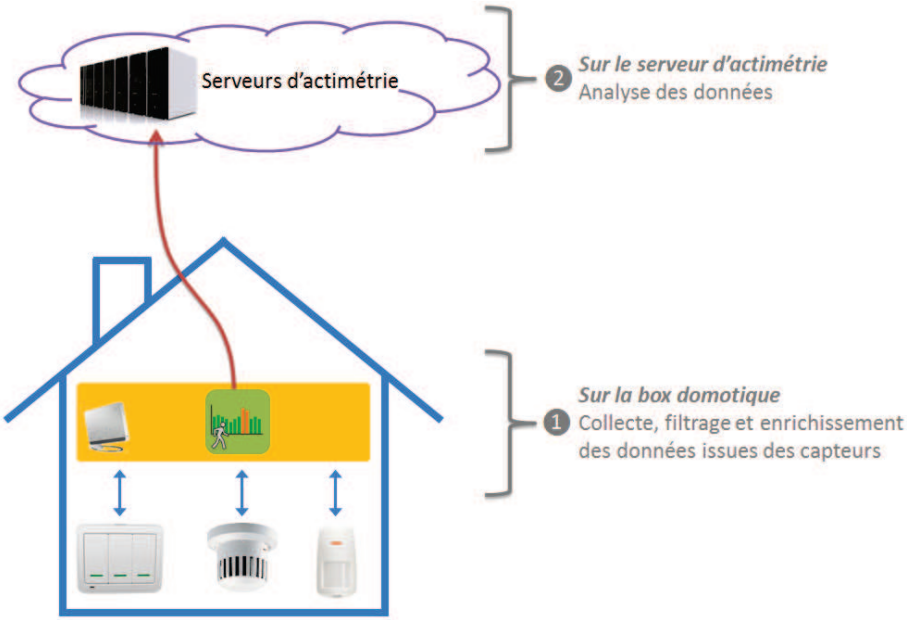

FIGURE 7.3 – Les deux parties de l'application Actimétrie.

Dans le cadre de cette validation, l'application est séparée en deux parties : la collecte, le filtrage et l'enrichissement des données brutes issues des capteurs de l'habitat, d'une part, et l'analyse de ces données, d'autre part (figure 7.3). La première partie de l'application est localisée sur la *home gateway*. C'est celle-ci qui fera l'objet de notre attention par la suite. L'analyse des données est, quant à elle, effectuée par un serveur distant. Pour nos travaux, nous avons réutilisé un composant logiciel développé par Orange dans le projet MIDAS. Celui-ci permet de calculer le taux d'occupation des pièces d'un habitat au jour le jour et de comparer celui-ci avec la référence (capture d'écran figure 7.4).

15. Présentation effectuée lors de l'Université d'été de la e-santé 2012 : `http://www.youtube.com/watch?v=2RmKvtCuy8I`

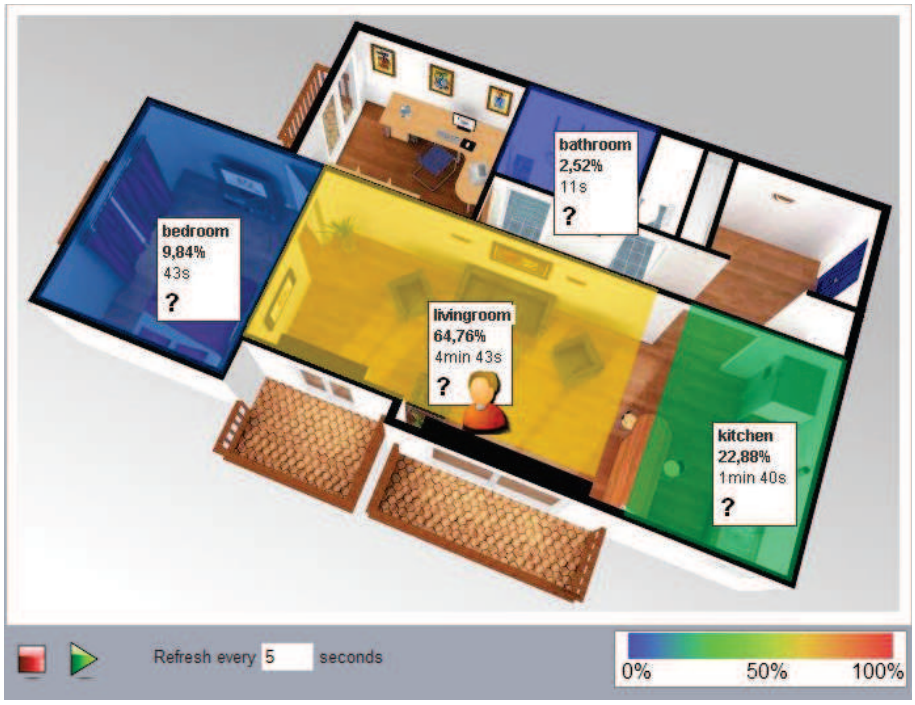

FIGURE 7.4 – Capture d'écran du tableau de bord de l'application d'actimétrie.

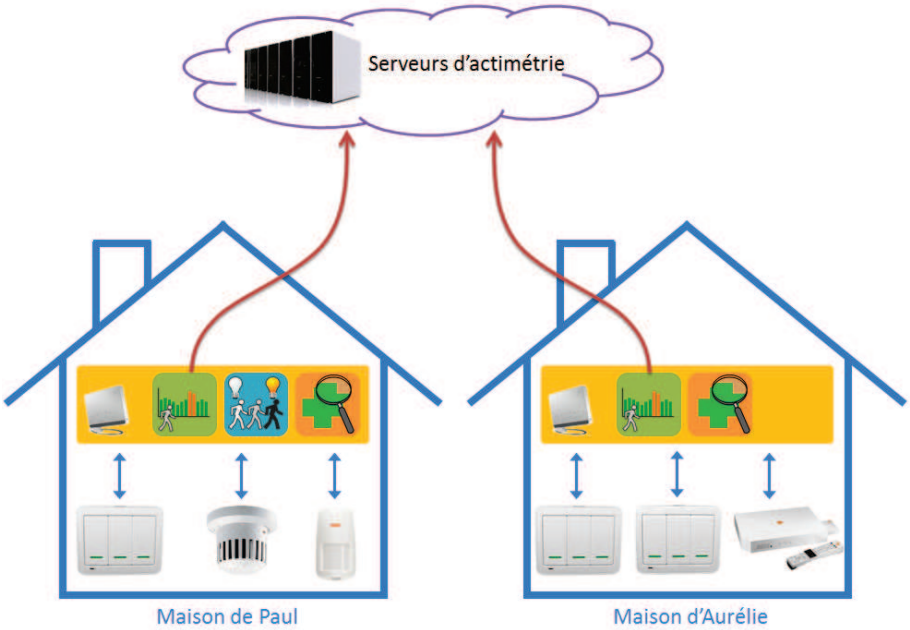

FIGURE 7.5 – Déploiement du service Actimétrie dans plusieurs habitats.

Pour terminer cette présentation du service Actimétrie, nous allons expliquer la manière avec laquelle il est déployé sur un ensemble d'habitats (figure 7.5). Tout d'abord, il faut souligner que chaque habitat dispose de son propre écosystème d'équipements. Le service devra donc être en mesure d'exploiter les équipements spécifiques à chaque habitant, tout en prévoyant la possibilité que ceux-ci tombent en panne ou que d'autres équipements soient ajoutés durant l'exécution. Notons, que l'application Actimétrie peut par ailleurs ne pas être la seule à s'exécuter sur la *home gateway*. Concernant le serveur d'Actimétrie MIDAS, celui-ci sera partagé entre les différents clients. Aussi, les données envoyées par chaque utilisateur devront au minimum être estampillées pour retrouver leur propriétaire, ainsi que pour des raisons de sécurité.

Maintenant que nous avons présenté le service Actimétrie dans son ensemble, nous allons focaliser notre attention sur la partie du service qui est exécutée sur la *home gateway*. En effet, notre validation s'intéresse exclusivement à la partie de l'application qui est exécutée dans la maison de l'usager, le reste relevant de la problématique du traitement des données dans le cloud. Et pour cela, nous allons commencer par exposer l'architecture du service chez le client.

7.2.2 Architecture du service sur la *home gateway*

Nous avons vu que le code de l'application Actimétrie déployé sur la *home gateway* a pour but de collecter les données à partir des équipements, de filtrer celles qui sont peu fiables, de les enrichir et de les transformer de manière à pouvoir les envoyer au serveur MIDAS. Nous sommes donc en présence d'une application de médiation, qui va transformer de manière incrémentale les données brutes des capteurs en données métier pour le serveur distant. Nous allons illustrer par un exemple les traitements successifs qui sont appliqués aux données, en présentant la médiation qui est déployée dans la maison d'Aurélie.

La médiation peut être divisée en deux parties (figure 7.6) : celle qui est effectuée directement par la plate-forme iCASA et celle qui est prise en charge par l'application Actimétrie. Grâce au *middleware* RoSe, la plate-forme gère l'accès aux équipements et met à disposition du programmeur des *proxies* de haut niveau. Leur cycle de vie suit celui des équipements : un *proxy* est créé quand un équipement est détecté et il est détruit quand l'équipement disparaît. Pour l'application, il s'agit du seul point d'accès aux équipements.

Côté application, les nœuds de médiation *stb* et *poussoir* ont pour rôle de récupérer les données en provenance des équipements. Comme leur nom l'indique, chaque nœud ne peut gérer qu'un type d'équipement particulier. Le nœud de médiation *stb* est donc lié au *proxy* de l'unique *set-top-box*, et le nœud de médiation *poussoir* est lié aux deux *proxyies* qui abstraient les deux boutons poussoir.

Une fois les données collectées, celles-ci sont transmises à un tronc commun dont le premier nœud de médiation va ajouter une localisation à chaque donnée. Pour cela, il fait appel à un service offert par iCASA qui permet de connaître la localisation de chaque équipement. Les données sont ensuite transmises à un médiateur *filtrage*, qui va éliminer les données pour lesquelles la localisation n'a pu être établie. Ce nœud va aussi supprimer les données dont

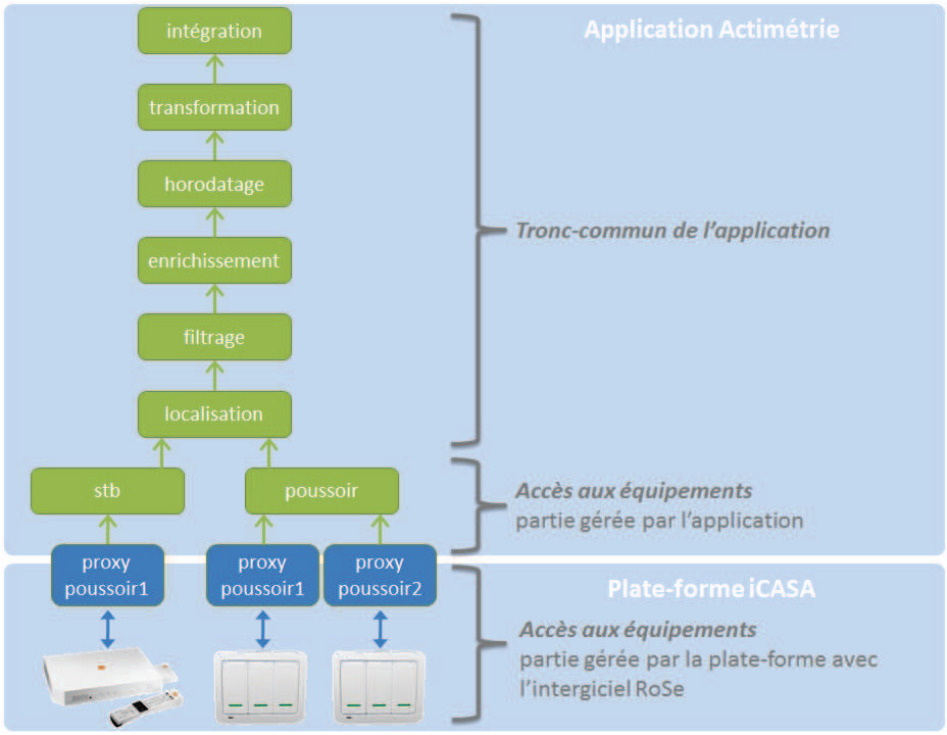

FIGURE 7.6 – Médiation de l'Actimétrie déployée dans la maison d'Aurélie.

la fiabilité de la mesure est jugée insatisfaisante. Le nœud suivant de la chaîne de traitement va enrichir la donnée en lui adjoignant l'identifiant de l'utilisateur. Ensuite, la donnée est horodatée, avant d'être transformée dans un format qui pourra être compréhensible par le serveur MIDAS. Enfin, le dernier nœud se charge de l'intégration dans le serveur, en effectuant l'envoi à l'aide d'une API SOAP [16].

Nous connaissons à présent l'architecture statique du service, avec la partie de la médiation qui est effectuée directement par la plate-forme iCASA et celle qui est prise en charge par l'application Actimétrie. Nous allons maintenant nous intéresser à l'évolution du service, que ce soit lors de son déploiement ou de sa mise à jour.

16. http://www.w3.org/TR/soap/

7.2.3 Déploiement et mise à jour du service

Nous avons vu précédemment que l'écosystème d'équipements pouvait être différent d'un habitat à l'autre. Aussi, la médiation de données effectuée chez le client devra être adaptée en conséquence. Pour cela, le déploiement de l'application sera effectué en plusieurs étapes (figure 7.7). Tout d'abord, le tronc commun de l'application sera installé chez l'usager ❶. En effet, les nœuds de médiation qui le composent seront utilisés quels que soient les équipements disponibles. Dans un second temps, les morceaux de médiation spécifiques aux équipements disponibles seront ajoutés ❷. Cela permettra de ne déployer et d'installer que le code nécessaire dans chaque habitat. Ceci fait, les *proxies* RoSe seront connectés à la chaîne de médiation ❸. Enfin, l'architecture globale de l'application pourra être validée, afin de diagnostiquer des erreurs lors du processus d'installation ❹.

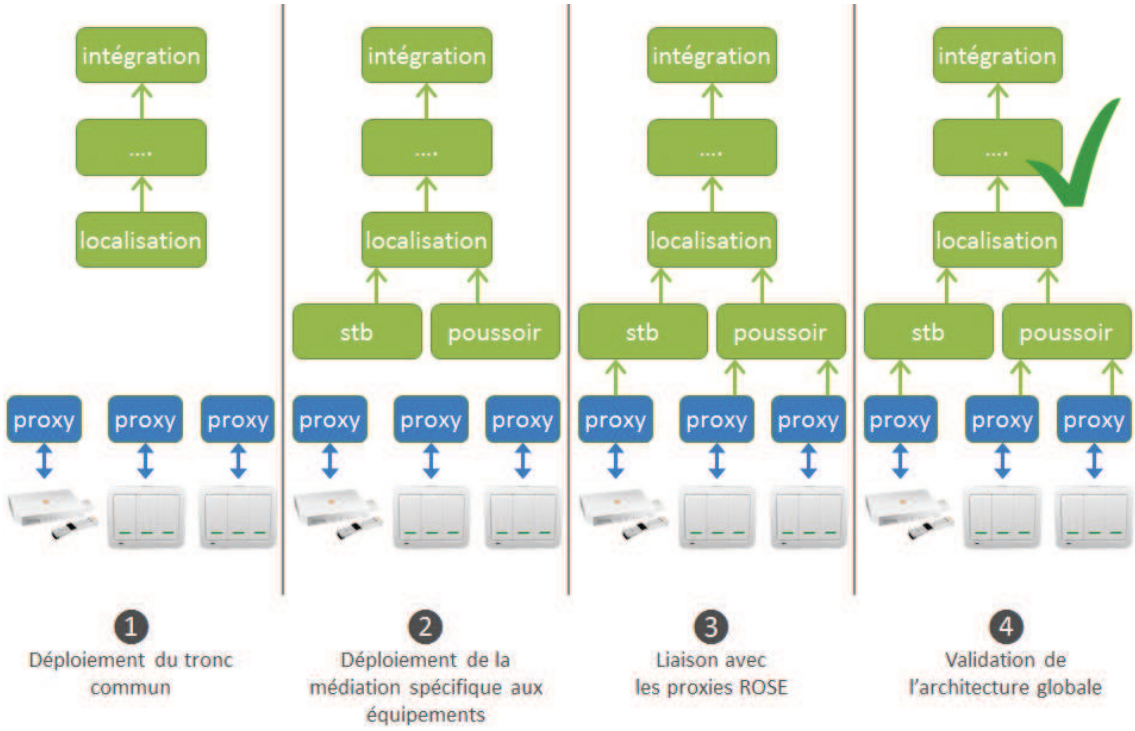

FIGURE 7.7 – Etapes du déploiement du service Actimétrie.

Concernant la mise à jour, deux cas de figure peuvent se produire : un nouvel équipement peut devenir disponible ou, au contraire, ne plus être accessible par l'application.

Dans le premier cas, si un équipement du même type est déjà installé sur la plate-forme, il suffira de lier l'application au *proxy* fourni par RoSe à l'aide du code déjà déployé. Dans le cas contraire, il sera nécessaire de télécharger et d'installer le morceau de médiation nécessaire à la communication de l'application avec le nouvel équipement, de lier le *proxy* RoSe à la chaîne de médiation augmentée par cette mise à jour et de valider l'architecture globale. Cela revient donc à effectuer les étapes ❷, ❸ et ❹ présentées ci-dessus.

7.2.4 Synthèse

Nous venons de présenter Actimétrie, un service du *Smart Home* qui a pour objet la détection de signes avant-coureurs de maladies invalidantes. Pour cela, le service analyse les modifications du rythme de vie d'une personne. En effet, plusieurs études ont montré que ces modifications d'habitudes sont des signaux faibles mais fiables précédant l'apparition de maladies invalidantes.

Pour effectuer son travail d'analyse, l'application Actimétrie est composée de deux parties. La première, située dans l'habitat collecte des signes d'activité du résident à partir d'un ensemble d'équipements : boutons poussoirs, *set-top-box*, détecteurs de présence. Ces données sont centralisées, filtrées et transformées sur la *home gateway*. Elles sont ensuite transmises à la seconde partie de l'application, qui est exécutée au niveau des serveurs du fournisseur de l'application Actimétrie. C'est là que les données sont analysées afin de détecter une éventuelle modification d'activité dans l'habitat.

Notre validation va se focaliser sur la partie hébergée sur la *home gateway* de l'usager. Ce morceau d'application doit être conçu de manière modulaire, en intégrant de la variabilité afin de ne déployer sur chaque *home gateway* que le code nécessaire à la prise en charge des équipements présents dans l'habitat. Par exemple, le morceau d'application qui permet la gestion des détecteurs de présence ne sera pas déployé si l'habitat n'est équipé qu'avec des détecteurs de mouvements. De plus, la plate-forme devra être dynamique ; c'est-à-dire permettre la mise à jour de l'application lors de son exécution. Pour reprendre l'exemple précédent, au branchement d'un détecteur de présence, l'application devra être augmentée de manière transparente du morceau de code qui permet la prise en charge de ce nouvel équipement.

Dans la partie qui va suivre, nous allons nous intéresser au développement et à la supervision du service Actimétrie. Nous allons présenter Cilia IDE, un atelier développé durant cette thèse qui permet de suivre le cycle de vie des chaînes de médiation Cilia, de leur conception jusqu'à leur exécution.

7.3 Développement et supervision du service Actimétrie avec Cilia IDE

Pour décrire l'utilisation de Cilia IDE dans le cadre du développement du service Actimétrie, nous allons considérer successivement l'élaboration de l'architecture de conception puis de l'architecture de déploiement. Ensuite, nous verrons le déploiement et la supervision de l'application. La description des gestionnaires autonomiques présents à l'exécution sera effectuée dans la section distincte qui suivra.

7.3.1 Architecture de conception

L'architecture de conception est définie sous la forme d'un fichier XML. Dans notre atelier Cilia IDE, ce dernier est réifié et présenté sous la forme d'un graphe (figure 7.8).

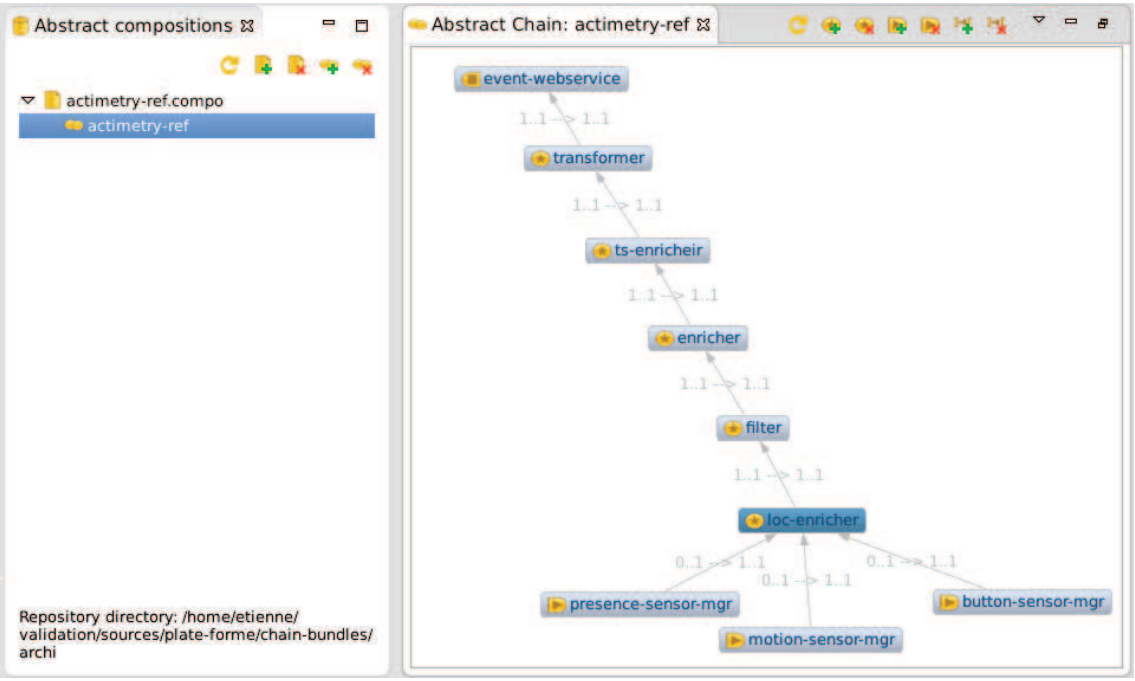

FIGURE 7.8 – Représentation de l'architecture de conception du service Actimétrie.

Nous retrouvons bien les cinq médiateurs et l'adaptateur de sortie de notre tronc commun, ainsi que les adaptateurs d'entrée qui permettent de gérer les équipements. Le médiateur *loc-enricher* est représenté dans une couleur différente des autres éléments de médiation car il s'agit d'une spécification. En effet, plusieurs stratégies de calcul de la localisation ont été élaborées et utiliser une spécification permettra de passer de l'une à l'autre à l'exécution tout en restant conforme à l'architecture de conception. Les liaisons présentent des cardinalités que nous décrirons par la suite.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <cilia-composition-specifications>
3    <chain id="actimetry-ref" namespace="fr.liglab.adele.cilia">
4      <adapters>
5        <adapter-implem id="event-webservice" type="EventServiceAdapter"/>
6        <adapter-implem id="button-sensor-mgr" type="number-generator-
           adapter"/>
7        <adapter-implem id="presence-sensor-mgr" type="number-generator-
           adapter"/>
8        <adapter-implem id="motion-sensor-mgr" type="number-generator-
           adapter"/>
9      </adapters>
10
11     <mediators>
12       <mediator-implem id="transformer" type="MeasureTransformerMediator"
           />
13       <mediator-implem id="ts-enricheir" type="TimeStampEnricherMediator"
           />
14       <mediator-implem id="enricher" type="MeasureEnricherMediator"/>
15       <mediator-implem id="filter" type="MeasureFilterMediator"/>
16       <mediator-specification id="loc-enricher" type="
           LocationEnricherSpec"/>
17     </mediators>
18
19     <bindings>
20       <binding from="loc-enricher:out" from-cardinality="1..1"
21                to="filter:in" to-cardinality="1..1"/>
22       <binding from="filter:out" from-cardinality="1..1"
23                to="enricher:in" to-cardinality="1..1"/>
24       <binding from="enricher:out" from-cardinality="1..1"
25                to="ts-enricheir:in" to-cardinality="1..1"/>
26       <binding from="ts-enricheir:out" from-cardinality="1..1"
27                to="transformer:in" to-cardinality="1..1"/>
28       <binding from="transformer:out" from-cardinality="1..1"
29                to="event-webservice:in" to-cardinality="1..1"/>
30       <binding from="presence-sensor-mgr:unique" from-cardinality="0..1"
31                to="loc-enricher:in" to-cardinality="1..1"/>
32       <binding from="button-sensor-mgr:unique" from-cardinality="0..1"
33                to="loc-enricher:in" to-cardinality="1..1"/>
34       <binding from="motion-sensor-mgr:unique" from-cardinality="0..1"
35                to="loc-enricher:in" to-cardinality="1..1"/>
36     </bindings>
37   </chain>
38 </cilia-composition-specifications>
```

Code source 7.1 – Fichier XML de l'architecture de conception du service Actimétrie.

Le Code source 7.1 présente le fichier XML à l'origine de la réification. Chaque fichier peut en fait définir plusieurs chaînes. Dans notre cas, il ne contient qu'une unique description qui porte l'identifiant *actimetry-ref*. La définition de la chaîne est effectuée en trois sections : la première concerne les adaptateurs, la seconde les médiateurs et la troisième les *bindings*.

Les adaptateurs définis sont au nombre de quatre : l'adaptateur de sortie et les trois adaptateurs d'entrée. Chacun d'eux dispose d'un identifiant unique et référence une implantation à l'aide des attributs *type* et *namespace*. De la même manière, les médiateurs sont définis à l'aide d'un identifiant et de la référence vers une implantation. Notons que dans notre exemple, nous référençons quatre implantations et une spécification (*loc-enricher*). Enfin, les *bindings* permettent de lier les éléments de médiation. Pour cela, ils définissent une source et une destination (*from* et *to*) avec les ports auxquels le *binding* peut être connecté. De plus, des cardinalités permettent d'exprimer de la variabilité. Dans notre cas, ils permettent de rendre les gestionnaires d'équipements optionnels.

7.3.2 Architecture de déploiement

Après avoir considéré l'architecture de conception, intéressons nous à l'architecture de déploiement. La figure 7.9 montre sa réification dans Cilia IDE.

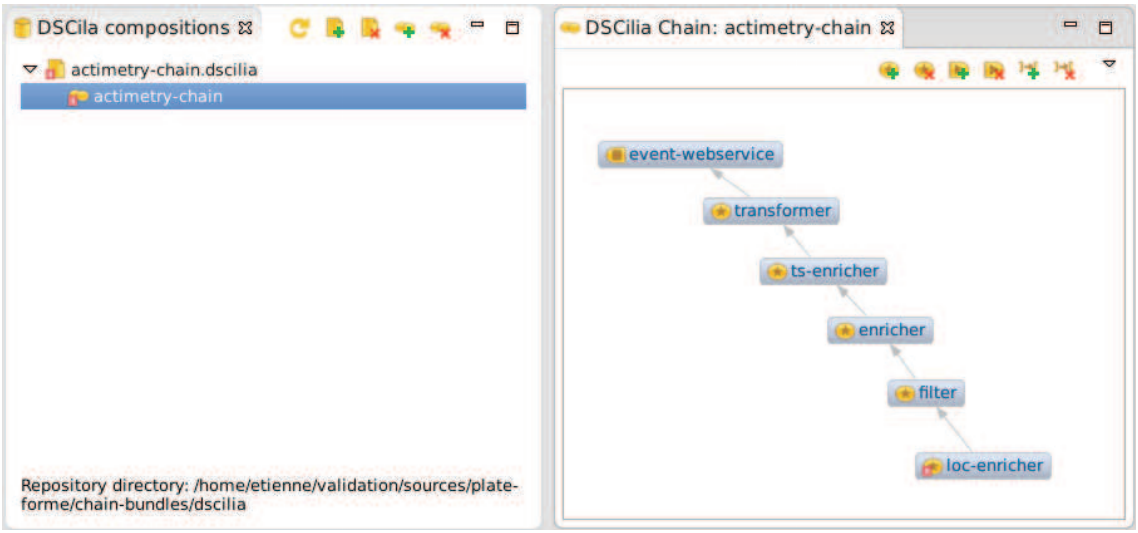

FIGURE 7.9 – Représentation de l'architecture de déploiement du service Actimétrie.

L'architecture de déploiement n'est constituée que du tronc commun (figure 7.6 page 181). Nous pouvons constater que le médiateur *loc-enricher* est marqué avec une erreur. Cela est normal car ce médiateur n'a pas de *binding* entrant. A l'exécution, cette architecture sera donc, elle aussi, en erreur pour la même raison. Cependant, nous verrons qu'un gestionnaire autonomique viendra compléter la chaîne Cilia afin de la rendre valide sitôt son déploiement terminé.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <cilia>
3    <chain id="actimetry-chain">
4      <adapters>
5        <adapter-instance id="event-webservice" type="EventServiceAdapter"
           />
6      </adapters>
7
8      <mediators>
9        <mediator-instance id="filter" type="MeasureFilterMediator" />
10       <mediator-instance id="enricher" type="MeasureEnricherMediator">
11         <processor>
12           <property name="user" value="Aurelie"/>
13         </processor>
14       </mediator-instance>
15       <mediator-instance id="ts-enricher" type="TimeStampEnricherMediator
           " />
16       <mediator-instance id="loc-enricher" type="LocationEnricherMediator
           " />
17       <mediator-instance id="transformer" type="
           MeasureTransformerMediator">
18     </mediators>
19
20     <bindings>
21       <binding from="loc-enricher:out" to="filter:in"/>
22       <binding from="filter:out" to="enricher:in"/>
23       <binding from="enricher:out" to="ts-enricher:in"/>
24       <binding from="ts-enricher:out" to="transformer:in"/>
25       <binding from="transformer:out" to="event-webservice:in"/>
26     </bindings>
27   </chain>
28 </cilia>
```

Code source 7.2 – Fichier XML de l'architecture de déploiement du service Actimétrie.

La syntaxe du fichier XML de l'architecture de déploiement (Code source 7.2) est assez proche de celle de l'architecture de conception, de part la proximité de leurs méta-modèles. Il existe tout de même des différences importantes :

- cette description d'architecture ne contient que des liens vers des implantations. Les liens vers des spécifications ont donc été substitués (comme par exemple dans le cas du médiateur *loc-enricher*) ;
- les cardinalités sont inexistantes sur les *bindings*, ce qui équivaut à des cardinalités *1..1* à chacune de leurs deux extrémités ;
- des informations de configuration peuvent être ajoutées.

7.3.3 Déploiement et supervision du service

Maintenant que l'architecture d'exécution est créée, nous allons détailler les étapes qui permettent de l'exécuter et de la superviser sur une plate-forme distante.

Définition de la plate-forme

Il est tout d'abord nécessaire de définir l'adresse de la plate-forme ainsi que le port sur lequel il sera possible de la contacter. Un bouton de la vue *Platforms* permet d'afficher une boîte de dialogue dans laquelle nous pouvons renseigner ces informations (figure 7.10).

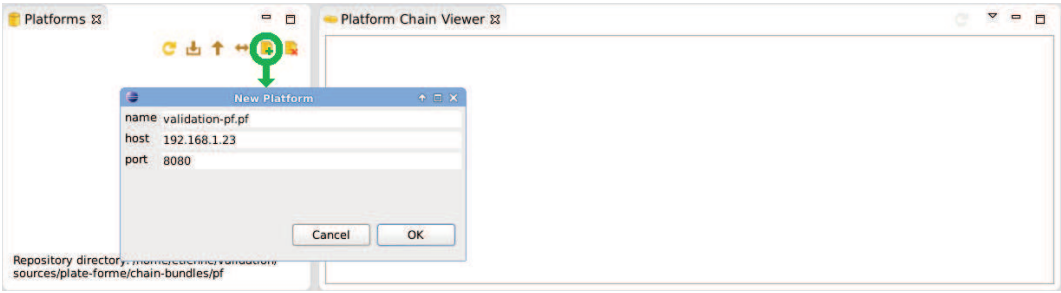

FIGURE 7.10 – Définition de l'adresse de la plate-forme distante dans Cilia IDE.

Transfert des artéfacts vers la plate-forme d'exécution

A présent, nous pouvons transférer l'ensemble des fichiers nécessaires vers la plate-forme distante. Pour cela, après avoir sélectionné la plate-forme précédemment créée, un second bouton permet de choisir la chaîne de médiation Cilia que l'on souhaite transférer. Dans notre cas, une seule architecture de déploiement a été définie dans l'atelier ; seule celle-ci est donc proposée (figure 7.11).

Après validation de la boîte de dialogue, l'atelier va regarder le contenu de la chaîne, afin de lister l'ensemble des ressources auxquelles elle fait référence. Le fichier dscilia, qui décrit l'architecture, et l'ensemble des fichiers JAR des implantations sont alors rassemblés dans un *deployment package* (fichier ZIP contenant un fichier manifest, compréhensible par OSGi™). Ce fichier est ensuite transmis à la plate-forme à l'aide d'une requête REST. Côté plate-forme distante, une fois le fichier reçu, celui-ci est déployé en utilisant les API d'OSGi™.

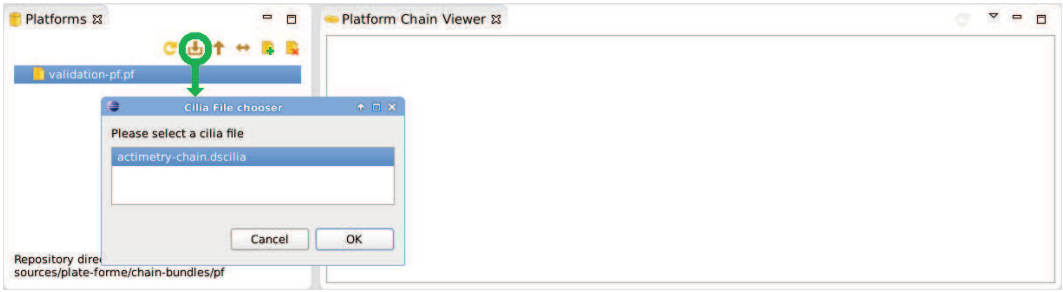

FIGURE 7.11 – Choix du fichier dscilia à transférer.

Visualisation de la chaîne déployée et analyse de l'état des éléments de médiation

Un autre bouton de la vue *Platforms* permet de contacter la plate-forme distante pour lui demander l'ensemble des chaînes de médiation qui sont exécutées (figure 7.12). Dans notre cas, seule la chaîne précédemment déployée est en cours d'exécution. De plus, une visualisation graphique permet de montrer l'architecture des chaînes exécutées.

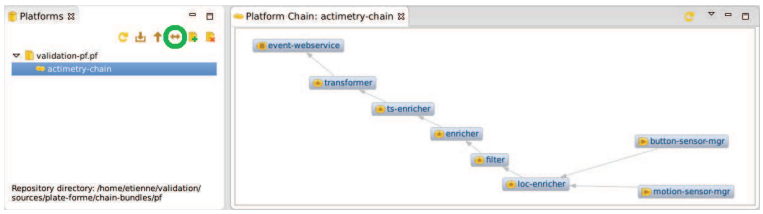

FIGURE 7.12 – Visualisation de la chaîne déployée avec Cilia IDE.

En double cliquant sur un élément de la chaîne, on accède à des informations détaillées, qui sont récupérées de manière transparente par notre atelier. Dans l'exemple de la figure 7.13, nous visualisons les informations générales relatives au médiateur de filtrage. Cela permet, en particulier, de constater que cet élément est dans un état valide. Notons que dans le cas contraire, une erreur aurait été reportée dans notre atelier.

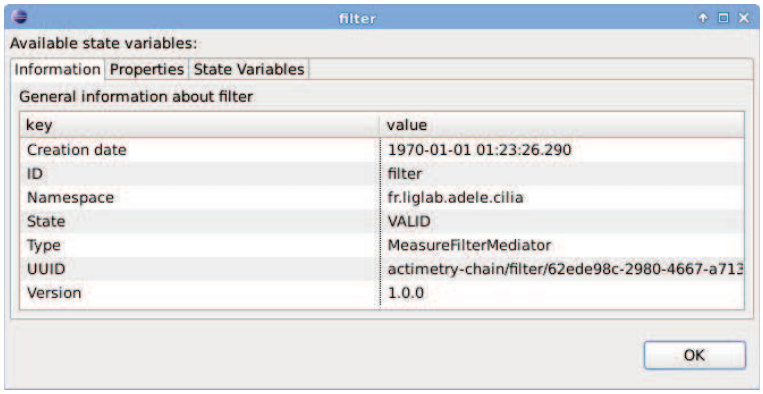

FIGURE 7.13 – Informations générales du médiateur de filtrage.

Le second onglet de la boîte de dialogue liste les propriétés (au sens iPOJO) de l'élément de médiation (figure 7.14). Ces propriétés peuvent être utilisées afin d'implanter des paramètres.

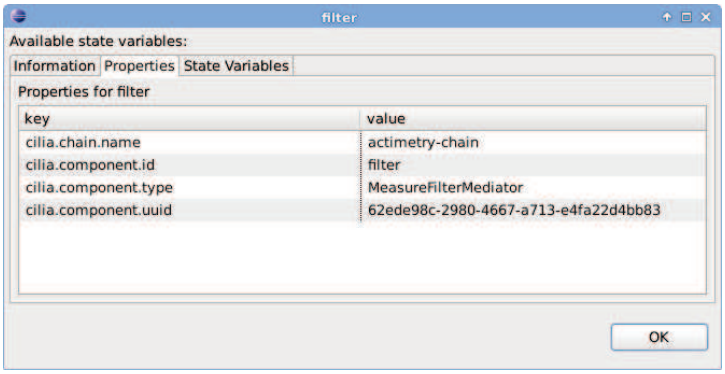

FIGURE 7.14 – Propriétés du médiateur de filtrage.

Enfin, un dernier onglet permet de connaître la liste des variables d'états (figure 7.15). A gauche de leur nom, une case à cocher permet d'activer chacune au cas par cas. Cela permet de ne collecter que les informations utiles à l'administration de la chaîne de médiation. A droite de leur nom, la valeur courante est affichée, si la variable d'états est activée. Enfin, pour rafraîchir à la demande les valeurs des variables, un bouton est disposé en bas de la boîte de dialogue.

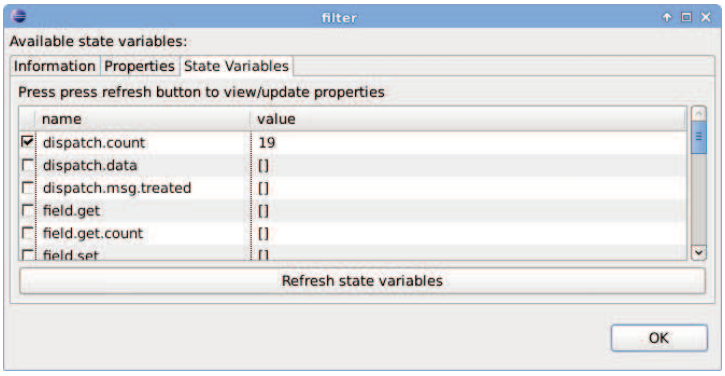

FIGURE 7.15 – Variables d'états du médiateur de filtrage.

Liaison avec l'architecture de conception et diagnostique des erreurs de conformité

Maintenant que nous connaissons avec précision ce qui est exécuté sur la plate-forme distante, il faut vérifier que cette exécution est conforme avec les décisions de conception rassemblées dans l'architecture éponyme. Pour cela, un bouton de l'atelier permet d'indiquer le lien entre ces deux architectures (figure 7.16). En cliquant sur celui-ci, une fenêtre permet la sélection de l'architecture de conception parmi toutes celles chargées dans notre atelier. Une fois le lien entre les architectures établi, l'atelier valide l'architecture exécutée à chaque mise à jour de celle-ci.

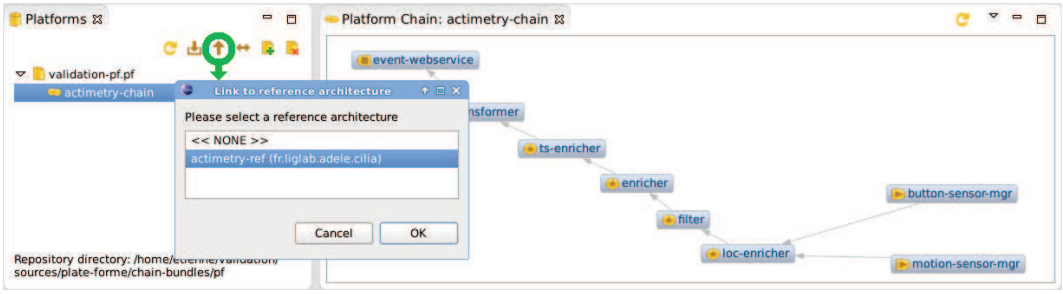

FIGURE 7.16 – Mise en relation de l'architecture d'exécution avec l'architecture de conception.

Pour illustrer la détection des erreurs de validité de l'architecture exécutée, nous avons créé sur la plate-forme d'exécution un *binding* entre le médiateur de filtrage et celui qui ajoute un horodatage. Ce *binding* est identifié par l'atelier comme étant invalide (figure 7.17) : il est affiché en rouge sur l'architecture exécutée et une erreur est affichée dans la vue qui centralise toutes les erreurs. Pour cela, l'atelier a exécuté en arrière-plan l'algorithme de validation que nous avons présenté dans la proposition.

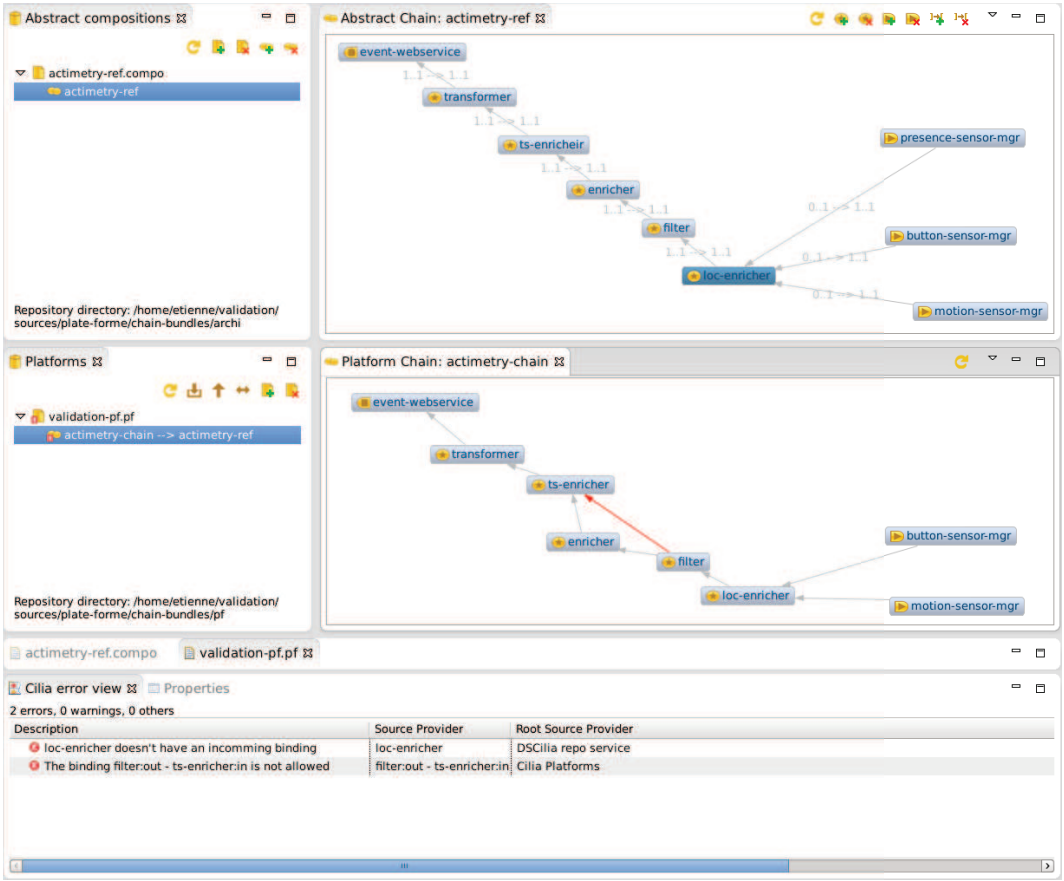

FIGURE 7.17 – Détection d'erreur de validation dans Cilia IDE.

De plus, nous avons vu que cet algorithme calculait la correspondance entre les composants de l'architecture de conception et ceux de l'architecture exécutée. Nous utilisons ce résultat intermédiaire pour que toute sélection dans la fenêtre d'une des deux architectures entraîne la coloration des éléments correspondants dans l'autre architecture. Dans l'exemple de la figure 7.18, le lien semble évident pour le développeur. Mais, dans des cas plus complexes, cette fonctionnalité se révèle extrêmement précieuse.

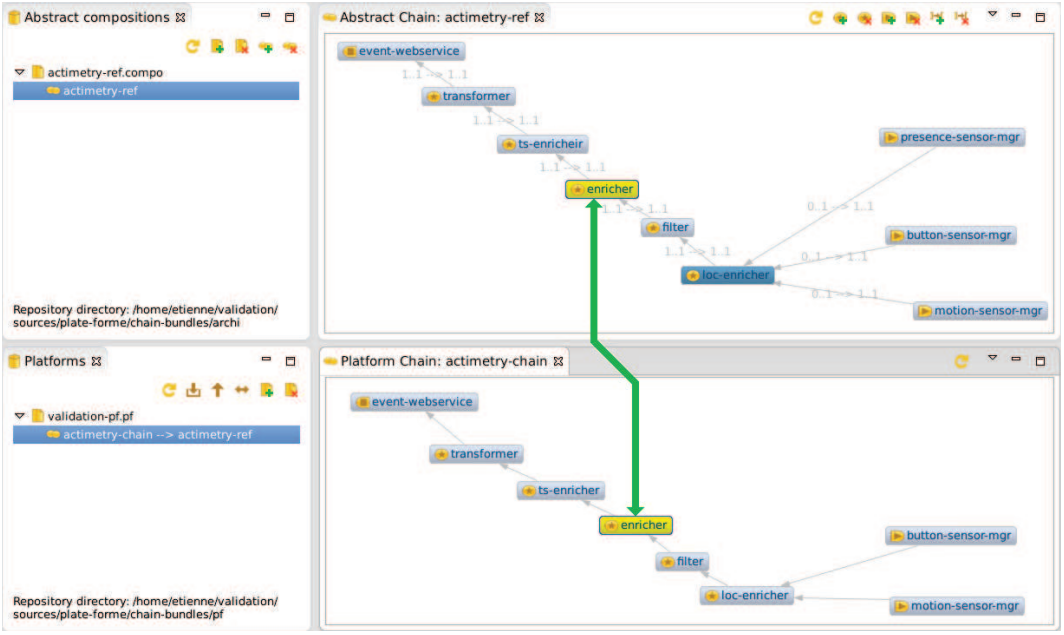

FIGURE 7.18 – Visualisation du lien entre les architectures dans Cilia IDE.

7.3.4 Synthèse

Cilia IDE est un atelier basé sur Eclipse qui permet de développer, de déployer et de superviser des chaînes de médiation Cilia. L'implantation de cet environnement a permis d'affiner et de valider la formalisation des composants et des architectures que nous avons présentés dans le chapitre proposition.

Nous avons illustré le fonctionnement de Cilia IDE avec le développement et la supervision du service Actimétrie. Nous avons vu que l'atelier permet d'utiliser des spécifications et des implantations de composants de médiation Cilia, présents dans la base de connaissances.

A partir de ceux-ci, Cilia IDE offre la possibilité de définir des architectures de conception (avec des cardinalités et des spécifications de composants) et des architectures de déploiement (sans variabilité). En définissant l'adresse d'une plate-forme d'exécution distante, il est possible de transférer une à une les architectures de déploiement. L'atelier se charge alors de rassembler tous les éléments nécessaires dans un *deployment package* et de transmettre celui-ci à la plate-forme distante. De son côté, cette dernière prend en charge l'initialisation de la chaîne de médiation Cilia reçue de l'atelier.

La chaîne exécutée à distance peut être supervisée directement dans l'atelier. Il est alors possible de vérifier l'état des éléments de médiation et de s'assurer que la chaîne Cilia est cohérente avec son architecture de conception. L'atelier utilise pour cela l'algorithme de vérification détaillé dans notre proposition. Cela permet d'afficher une liste d'erreurs dans une vue de l'atelier en cas d'invalidité de l'architecture exécutée. Enfin, l'atelier utilise le résultat de l'algorithme de correspondance afin de montrer les liens entre les composants d'une architecture exécutée et ceux de son architecture de conception.

7.4 Administration autonomique

Nous venons de décrire comment créer, déployer et superviser une chaîne de médiation Cilia à l'aide de Cilia IDE. Cependant, nous avons vu en introduction qu'il était nécessaire d'automatiser les tâches d'administration dans le *Smart Home*, car il n'y a pas d'administrateur disponible lors de l'exécution des applications pervasives. C'est la raison pour laquelle nous avons recours à l'informatique autonomique.

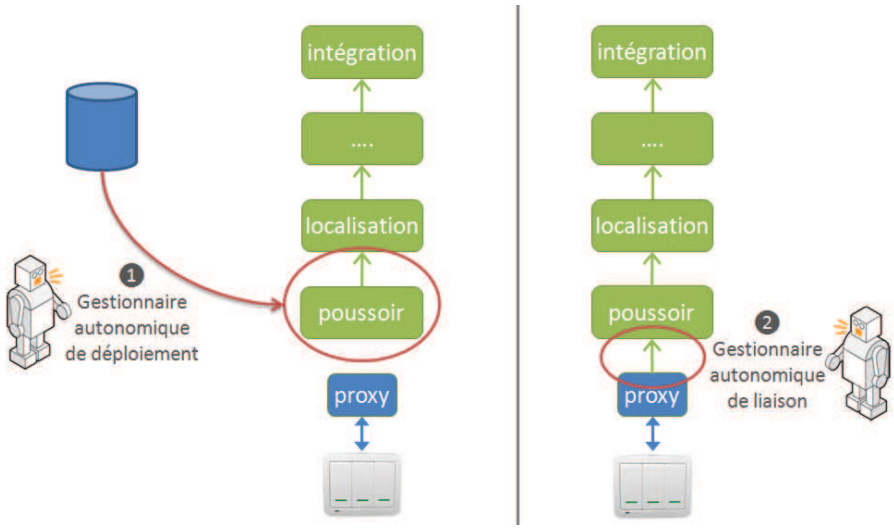

FIGURE 7.19 – Les deux gestionnaires autonomiques de notre validation.

Dans le cadre de l'application Actimétrie, nous avons utilisé deux gestionnaires autonomiques complémentaires (figure 7.19) : le premier ❶ permet de télécharger et installer sur la plate-forme le code nécessaire à l'utilisation des ressources disponibles dans le contexte d'exécution. Il s'agit du gestionnaire autonomique mentionné sur la figure 6.1 page 159. Le second ❷ est directement intégré à la plate-forme Cilia. Il s'occupe de lier les *proxies* RoSe aux adaptateurs d'entrée. Nous allons maintenant illustrer le fonctionnement du premier gestionnaire autonomique (celui que nous avons réalisé).

Le rôle du gestionnaire autonomique de déploiement est de compléter la chaîne de médiation en lui ajoutant des morceaux en fonction des ressources disponibles sur la plate-forme d'exécution. La figure 7.20 donne un exemple de l'action du gestionnaire autonomique de déploiement. Lors du démarrage de l'application, nous pouvons constater que la

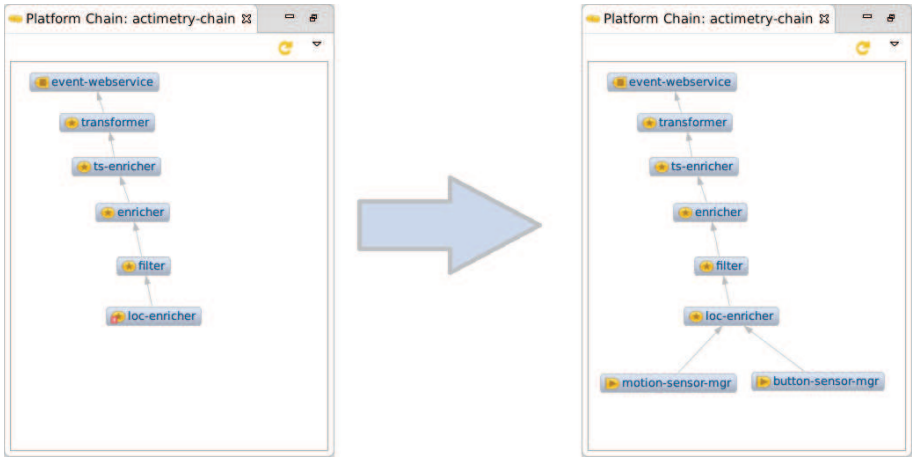

FIGURE 7.20 – Exemple de réification de la chaîne à l'exécution avant et après intervention du gestionnaire autonomique de déploiement.

chaîne est rigoureusement conforme à la description de l'architecture de déploiement (voir paragraphe précédent). Après intervention du gestionnaire autonomique, deux adaptateurs d'entrée ont été ajoutés afin de permettre au tronc commun de recevoir des mesures en provenance des détecteurs de mouvements et de boutons poussoirs.

La chaîne de médiation Cilia se trouve alors mise à jour et opérationnelle. Il ne reste plus alors qu'à exécuter l'algorithme de validation, afin de vérifier la conformité entre l'architecture de l'exécution et son architecture de conception.

7.5 Tests de performance

Pour terminer ce chapitre consacré à notre validation de thèse, nous allons évaluer le temps d'exécution de nos algorithmes. Pour cela, nous avons mis en place deux tests décrits dans les deux paragraphes suivants. Le premier calcule le temps d'exécution de l'algorithme sur une architecture de type chaîne, afin dévaluer les parcours de liaisons. Le second test considère une architecture sans liaison, afin d'évaluer le temps de calcul nécessaire à l'évaluation des correspondances de types. Un troisième paragraphe met en perspective les résultats obtenus au regard des temps de reconfiguration de la plate-forme à composants dynamiques Cilia.

7.5.1 Utilisation des liaisons avec une architecture de type chaîne

Le premier test que nous avons effectué, est un calcul de correspondance dans le cas d'une architecture de type chaîne. La figure 7.21 montre l'architecture de conception que nous avons utilisée, ainsi qu'un exemple d'architecture de l'exécution à cinq composants. Pour effectuer son calcul de correspondance, l'algorithme commence par s'appuyer sur un premier résultat. Par exemple, il pourra établir une correspondance entre b_1 et *confImplB*.

A la suite de cette première correspondance, le parcours des liaisons entrantes et sortantes permet de trouver les autres correspondances.

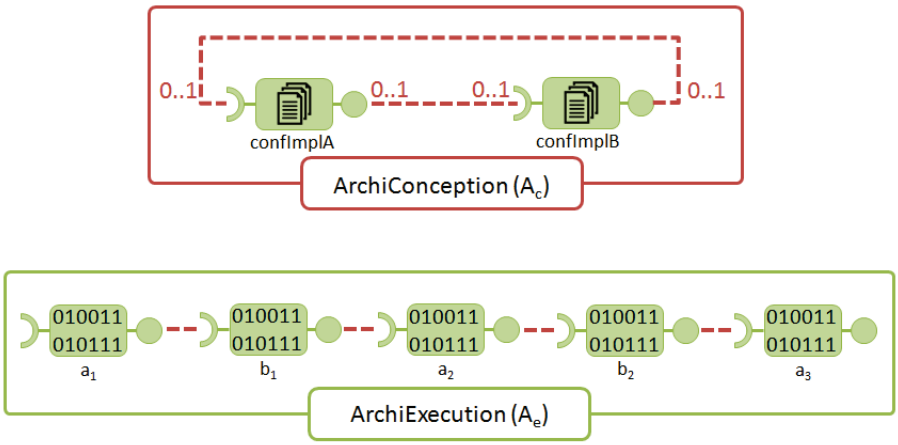

FIGURE 7.21 – Architecture de type chaîne.

La figure 7.22 montre l'évolution du temps de calcul des correspondances en fonction du nombre de composants de l'architecture de l'exécution. Nous pouvons constater que ce temps est pratiquement proportionnel au nombre de composants, ce qui est cohérent avec la complexité de notre algorithme de parcours des liaisons.

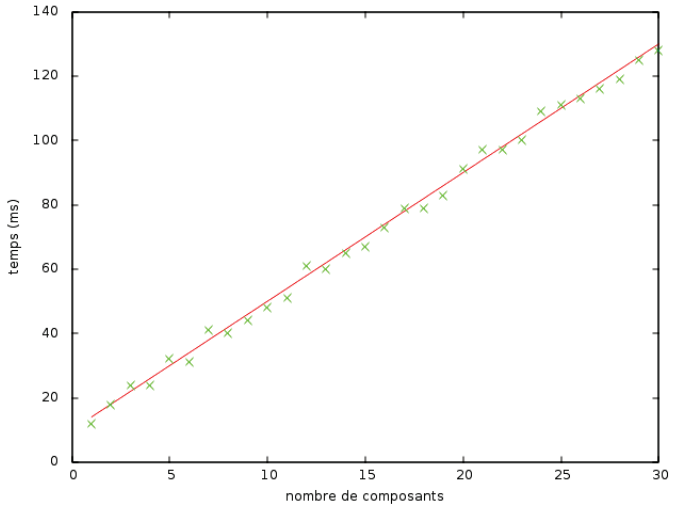

FIGURE 7.22 – Temps de calcul des correspondances dans une architecture de type chaîne.

7.5.2 Utilisation des types

Le second test met en oeuvre une architecture sans liaison (figure 7.23). Bien entendu, celle-ci est invalide, mais il est tout de même possible de calculer la correspondance entre les composants de l'architecture de conception et ceux de l'architecture de l'exécution.

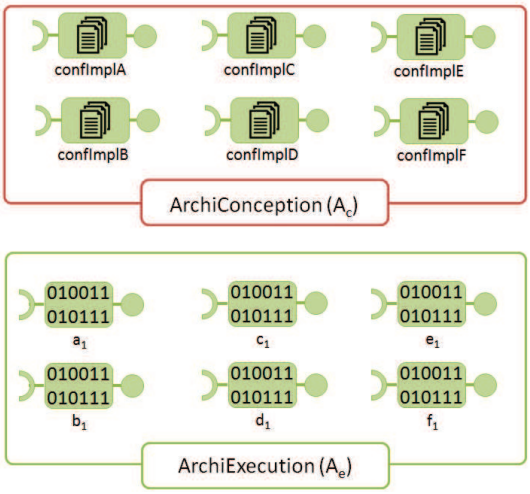

FIGURE 7.23 – Architecture sans liaison.

La figure 7.24 montre l'évolution du temps de calcul des correspondances en fonction du nombre de composants de l'architecture de l'exécution. Ici aussi, nous pouvons constater que ce temps est pratiquement proportionnel au nombre de composants, ce qui est cohérent avec notre algorithme dont la tâche la plus complexe est le calcul des structures d'initialisation, dont la complexité est fonction du nombre de composants de l'architecture de l'exécution (et indépendante du nombre de composants de l'architecture de conception).

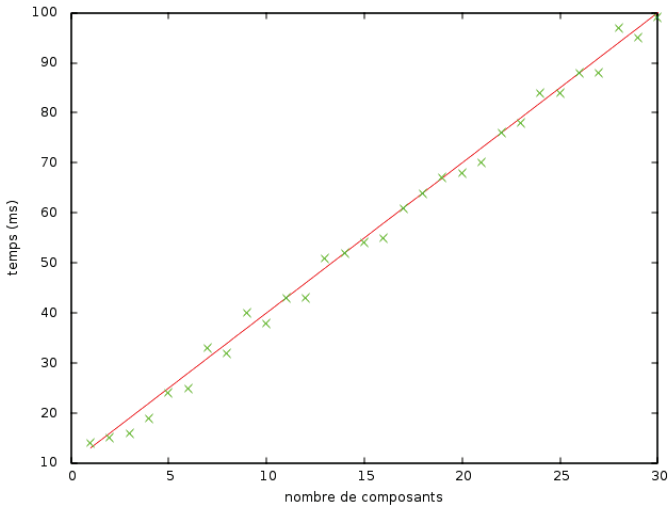

FIGURE 7.24 – Temps de calcul des correspondances dans une architecture sans liaison.

7.5.3 Interprétation des résultats obtenus

Les temps d'exécution, que nous venons de présenter, sont à mettre en perspective avec les temps de reconfigurations des différentes opérations prises en chage par la plate-forme Cilia. Ceux-ci ont fait l'objet de publications [GGMD+11, Mor13] et sont récapitulés dans le tableau 7.1.

OPÉRATIONS	TEMPS (MS)
Création d'un médiateur	130
Création des liaisons	60
Destruction des liaisons	50
Destruction d'un médiateur	115
Remplacement d'un médiateur	200
Reconfiguration d'un médiateur	7

Tableau 7.1 – Temps de reconfiguration de la plate-forme Cilia par opération [GGMD+11].

Nous pouvons constater que le délai induit par le remplacement d'un médiateur est très largement suppérieur à l'exécution de l'algorithme de validation sur une architecture de 30 composants. Aussi, dans la mesure où Cilia a été pensé pour des des architectures de médiation ne présentant au plus que quelques dizaines de médiateurs, les temps nécessaires à l'exécution de nos algorithmes restent tout à fait acceptables. De plus, nous traitons ici le cas d'une validation totale de l'architecture, alors que nous avons proposé des algorithmes qui permettent d'effectuer celle-ci de manière incrémentale.

Enfin, notons que si on souhaitait valider des architectures de très grande taille, il serait possible de diminuer significativement le temps d'exécution de nos algorithmes en effectuant l'étape d'initialisation elle aussi de manière incrémentale, ce qui n'est pas aujourd'hui le cas.

7.6 Conclusion

Notre validation de thèse a été effectuée dans le cadre du projet FUI MEDICAL. Ce projet collaboratif avait pour objectif de concevoir un *middleware* pour le *Smart Home* qui facilite le développement et l'administration d'applications intégrant des objets communicants hétérogènes (fournisseurs, protocoles de communication, schémas de données). Il a livré pour cela la plate-forme logicielle iCASA, un *middleware* embarqué dans la *home gateway* qui accueille les applications du *Smart Home*.

L'une de ces applications se nomme Actimétrie. Elle a pour but de favoriser la détection précoce de maladies invalidantes en analysant les habitudes d'une personne à son domicile. En effet, une modification du rythme de vie peut être un signal faible mais fiable de début de maladie invalidante. Cette application est composée de deux parties. La première, exécutée sur la *home gateway* de l'usager, s'occupe de la collecte, du filtrage et de la transformation

des données. Celles-ci sont ensuite envoyées à la seconde partie de l'application qui effectue l'analyse de ces données sur les serveurs du fournisseur du service Actimétrie. Pour cette validation, nous nous sommes focalisé sur la première partie.

Le code hébergé sur la *home gateway* a été conçu à l'aide du *framework* de médiation Cilia. Le code de l'application a donc été modularisé, ce qui permet de ne déployer dans l'habitat de chaque usager que les composants nécessaires à la gestion des équipements dont il dispose. En conséquence de cela, après l'installation d'un équipement d'un type nouveau dans l'habitat, il est nécessaire de déployer du code pour pouvoir communiquer avec celui-ci. L'application peut donc présenter des architectures variables au cours du temps, au fur et à mesure de l'enrichissement de l'écosystème d'équipements de l'usager. Notre travail de validation s'est donc intéressé au suivi et à la validation de ces architectures successives. Pour cela, nous avons présenté successivement un atelier de développement et de supervision nommé Cilia IDE et des gestionnaires autonomiques.

L'atelier Cilia IDE permet de définir des architectures de conception (avec de la variabilité) et de déploiement qui sont stockées dans la base de connaissances. Celle-ci contient de plus les réifications des architectures exécutées sur des plates-formes distantes. L'atelier utilise aussi les algorithmes de correspondance et de validation que nous avons définis dans notre proposition. L'algorithme de correspondance calcule le lien entre les composants d'une architecture de conception et ceux d'une architecture exécutée. Cela permet au développeur de mieux comprendre les relations entre ces deux architectures. L'algorithme de validation va un cran plus loin et vérifie la conformité entre les deux architectures. En cas de non validité, une liste d'erreurs est affichée dans l'atelier afin d'orienter le travail de l'administrateur. Cilia IDE est donc un puissant allié pour développer, déployer, tester et superviser des applications à base de composants Cilia. Cependant, dans le domaine du *Smart Home*, l'absence d'administrateur impose l'automatisation de la mise à jour des applications. C'est là l'objet de la seconde partie de la validation.

Les gestionnaires autonomiques que nous avons présentés automatisent l'adaptation de l'application à l'arrivée ou au départ d'équipements dans l'habitat. Le premier permet de télécharger et d'installer sur la plate-forme le code nécessaire à l'utilisation des équipements ; le second s'occupe de lier les composants exécutés aux équipements et de retirer ce lien si l'équipement devient indisponible. Ces gestionnaires autonomiques ont été implantés selon l'architecture MAPE-K proposée par IBM. Cela sépare de manière claire les différents traitements et les responsabilités associées à chaque tâche.

Au-delà des mesures brutes obtenues lors de l'évaluation des performances de nos algorithmes, il est important de constater que le temps de calcul évolue linéairement (et non exponentiellement) avec le nombre de composants de l'architecture. De plus, comme dans la pratique le nombre de composants est souvent inférieur à quelques dizaines, le temps de validation n'est pas supérieur au temps de remplacement d'un médiateur, par exemple. De plus, nous rappelons que notre expérimentation a porté sur la validation totale d'une architecture, alors que celle-ci sera effectuée de manière incrémentale dans la pratique. Nos algorithmes ont donc les performances requises pour les applications de médiation.

Quatrième partie

Conclusion et Perspectives

Chapitre 8

Conclusion et Perspectives

Sommaire

8.1 Conclusion **202**
8.1.1 Contexte 202
8.1.2 Besoins 203
8.1.3 Contributions 203
8.2 Perspectives **204**
8.2.1 Prolongements directs de notre validation 204
8.2.2 L'informatique autonomique 205
8.2.3 La modélisation du contexte 206

8.1 Conclusion

8.1.1 Contexte

Cette thèse se place dans le champ de l'**adaptation logicielle**. Dans ce cadre, nous avons souligné que la demande d'adaptation est de plus en plus présente dans les applications modernes afin de faire face à l'évolution des besoins, aux changements de disponibilité des ressources, à la nécessité de corriger les dysfonctionnements et à la mobilité des utilisateurs. De plus, cette contrainte est d'autant plus importante que les interruptions de services sont dans nombre d'applications mal acceptées, voire totalement inenvisageables au-delà de quelques secondes.

Pour répondre à ce besoin d'évolution lors de l'exécution, l'**informatique orientée service dynamique** est aujourd'hui largement utilisée. Elle propose un faible couplage qui autorise la substituabilité de modules logiciels appelés services. De plus, les liaisons retardées sont bien adaptées à des applications caractérisées par de fortes contraintes de dynamisme et une faible prédictibilité.

Le rapprochement de l'approche à services avec l'approche à composants a donné naissance à l'**approche à composants à services**. Cette nouvelle approche permet de faciliter la construction d'applications et leur adaptation à l'exécution en utilisant l'approche à services pour résoudre les dépendances entre les composants.

Pour automatiser ces adaptations lors de l'exécution, l'**informatique autonomique** a cherché des patrons afin de mettre en place des systèmes autogérés. C'est dans ce contexte qu'IBM a proposé l'architecture MAPE-K qui est aujourd'hui largement employée. Celle-ci sépare les préoccupations d'adaptation en quatre fonction qui partagent une **base de connaissance**. Cette dernière est d'une importance capitale. En effet, c'est à la lumière de cette connaissance que l'on peut comprendre et analyser le système afin de mettre en œuvre les stratégies d'adaptation.

Les technologies actuelles offrent donc les mécanismes nécessaires à l'adaptation d'applications de grande taille. Cependant, face à la complexité croissantes des applications, la seule satisfaction des dépendances de services au niveau des interfaces des composants à services n'est plus suffisante. Il est nécessaire de disposer d'une compréhension globale de l'application, afin de guider le mécanisme de reconfiguration.

Pour cela, le **niveau architectural** peut être extrêmement précieux. En effet, par la montée en abstraction qu'il propose, il permet de disposer d'une compréhension globale du système, que ce soit pour un administrateur humain ou pour un gestionnaire autonomique. C'est la raison pour laquelle de nombreuses approches se sont appuyées sur de telles abstractions depuis le début des années 2000, notamment dans le sillage des travaux de D. Garlan et S.-W. Cheng sur le *framework* Rainbow [Che08].

Cependant, nous avons vu que cette connaissance était bien souvent peu formalisée. C'est à cela que les **lignes de produits dynamiques** ont cherché à répondre en s'appuyant sur l'expertise acquise dans le cadre des lignes de produits traditionelles.

8.1.2 Besoins

Malheureusement, les solutions proposées sont restées relativement ad hoc et peu généralisable. **Or, s'il est évident que la connaissance propre à l'administration de chaque système doit être construite ou fournie au cas par cas, il serait tout de même judicieux d'offrir un canevas dans lequel celle-ci puisse facilement être intégrée.**

Conformément à notre état de l'art, nous pensons qu'un tel canevas devrait :

- s'appuyer à l'exécution sur des composants orientés services ;
- offrir une compréhension de cette exécution avec un point de vue architectural ;
- présenter les contraintes de conception, elles aussi au niveau architectural ;
- assurer une continuité de point de vue entre la conception et l'exécution [FR07] ;
- donner des algorithmes de vérification, afin de repérer des violations de contraintes de conception lors de l'exécution.

Plus précisément, il serait nécessaire d'offrir un canevas indépendant d'un modèle à composant particulier, qui puisse être facilement projeté sur tel ou tel *framework*.

8.1.3 Contributions

Pour atteindre nos objectifs, nous avons étudié le cycle de vie des architectures et des composants logiciels qui les composent. Cela nous a amené à définir un ensemble de méta-modèles architecturaux en relation :

- **le méta-modèle de conception** permet de définir une architecture qui rassemble toutes les décisions de conception d'une application, et de déterminer ainsi l'ensemble de ses exécutions valides ;
- **le méta-modèle de déploiement** offre la possibilité de décrire des configurations qui devront être instanciées au démarrage d'un système ;
- **le méta-modèle de l'exécution** permet de réifier les phénomènes d'exécution afin d'offrir un point de vue cohérent et de haut niveau sur l'exécution.

De plus, nous avons établi des liens entre ces différents méta-modèles, ce qui nous permet de disposer d'une continuité de point de vue depuis la conception jusqu'à l'exécution. En sens inverse, des algorithmes que nous avons spécifiés, nous permettent de vérifier qu'une architecture de l'exécution respecte les contraintes définies dans son architecture de conception. Pour mener à bien ce travail, il a aussi été nécessaire de clarifier la notion de composant. En effet, nous avons montré qu'il existe des différences conceptuelles importantes entre un composants spécifié, implanté et exécuté.

Cette recherche conceptuelle a été adossée de réalisations concrètes. Tout d'abord, un IDE a été développé afin de permettre la définition d'architectures de conception et de déploiement. Celles-ci peuvent être transférées sur une plate-forme d'exécution distante (qui

utilise *Cilia Mediation Framework*) qui transmet en retour à l'atelier les informations nécessaires au suivi de l'exécution.

Nous avons aussi mis en place un gestionnaire autonomique pour déployer de manière automatisée du code sur la plate-forme d'exécution.

Ces différentes réalisations ont été appliquées à l'informatique *pervasive* dans le cadre du projet FUI MEDICAL, sur un *middleware* pour le *Smart Home* appelé iCASA. Dans le cadre d'une application nommée actimétrie, nous avons utilisé l'atelier Cilia IDE pour concevoir, déployer et superviser une application de médiation de données. De plus, notre gestionnaire autonomique a été utilisé pour compléter l'application au fur et à mesure de l'arrivée de nouveaux dispositifs communiquants dans l'habitat de l'usager.

En focalisant notre travail sur les plates-formes à composants à services, nous avons donc effectué un pas significatif dans la compréhension et la réutilisation des infrastructures pour l'adaptation d'applications dynamiques. Nous avons ainsi fait un pas supplémentaire, là où la trop grande généricité des approches lignes de produits dynamiques peinaient à avancer.

8.2 Perspectives

Dans la direction tracée par notre travail, de nombreux prolongements seraient intéressants à investiguer. Nous avons regroupé ceux-ci en trois catégories : les prolongements directs de notre validation, ceux relatifs à l'informatique autonomique et la modélisation du contexte.

8.2.1 Prolongements directs de notre validation

Notre contribution a été validée par un atelier et un gestionnaire autonomique. De nombreux travaux restent encore à faire pour l'un comme pour l'autre.

L'atelier Cilia IDE

Concernant l'atelier, celui-ci permet aujourd'hui de superviser l'exécution, sans intervenir sur celle-ci après le déploiement. Il est clair que la modification de l'exécution est une fonctionnalité clé qu'il conviendrait d'implanter. Les opérations suivantes devraient être implantées de manière prioritaire :

- l'ajout et la suppression de liaisons ;
- l'ajout et la suppression de composants, ainsi que la modification de leur configuration ;
- la substitution de composant avec, potentiellement, le déploiement de nouvelles implantations.

Par ailleurs, en cas de violation de contrainte de l'architecture de conception, il peut arriver que ce soit l'architecture de conception elle-même qui doive être mise à jour pour prendre en considération davantage de cas. Un processus automatisé pourrait être envisagé pour cela.

Le gestionnaire autonomique

Côté gestionnaire autonomique, celui que nous avons implanté est relativement basique et s'il peut être paramétré pour d'autres cas d'usages, il faut reconnaître que son faible périmètre fonctionnel n'autorise qu'une réutilisation limitée. Son amélioration, notamment par des stratégies de correction avancées de l'architecture de l'exécution, pourrait faire l'objet d'une vaste étude, mettant en jeu la théorie des graphes.

Les solutions que nous avons présentées reposent sur un strict partage entre les décisions prises de manière automatisée et celles laissées à la charge d'un administrateur (primo-déploiement). Une plus grande flexibilité dans laquelle l'administrateur puisse revenir dans la boucle quand il le souhaite serait souhaitable.

8.2.2 L'informatique autonomique

Après avoir traité les prolongements les plus immédiats de nos travaux, nous allons nous intéresser au développement à plus long terme de l'informatique autonomique. Comme nous l'avons vu, ce domaine est encore relativement jeune et, malgré le grand nombre de travaux en cours, il reste encore de très nombreux défis à relever qui nécessitent un approfondissement des études existantes [Gar13]. Nous en avons sélectionné trois, qui nous semblent à la fois nécessaires et complexes à traiter.

Le premier défi est la prise en compte de l'incertitude. Celle-ci se situe à toutes les étapes du processus, depuis la remonté des mesures qui peuvent être entachées d'erreurs jusqu'à l'exécution de modifications, qui peut ne pas se dérouler de la manière souhaitée. Leur prise en compte au niveau de toutes les étapes de traitement du gestionnaire autonomique est un défi de taille.

Nous avons vu qu'un gestionnaire autonomique pouvait mettre en œuvre plusieurs types d'adaptations. Nous nous sommes intéressés aux adaptations réactives, qui visent à adapter au plus vite le système à une modification du contexte. Un autre type d'adaptation, dont l'exécution est potentiellement plus lente concerne l'optimisation du système. Ce processus peut mettre en œuvre une grande quantité de calculs, afin d'optimiser les traitements et la consommation des ressources. L'utilisation conjointe de ces deux approches (réactive et optimisation) au sein d'un même système mériterait une étude approfondie, en particulier, pour traiter les modifications concurrentes.

D'autres défis sont liés à la taille croissante des systèmes. En plus des adaptations concurrentes, que nous venons d'évoquer, l'adaptation de systèmes distribués est encore sous étudiée, même si certains travaux se sont déjà penchés sur cette question [Deb14].

Enfin, pour améliorer le processus d'adaptation, il pourrait être intéressant de découvrir des scénarios d'interaction qui structurent dans la pratique les demandes d'évolution.

Cela permettrait de formaliser un ensemble de chemins qui pourraient être utilisés à la fois pour rendre plus efficace le processus d'adaptation et pour préparer ou anticiper certaines reconfigurations.

8.2.3 La modélisation du contexte

Le dernier point que nous souhaiterions soulever, concerne la modélisation du contexte. Celle-ci est un enjeu clé, aussi bien dans le cadre de l'administration autonomique que dans celle mettant en jeu un administrateur humain. En effet, une compréhension fine du contexte d'exécution est absolument nécessaire afin de mettre en œuvre les adaptations les plus pertinentes.

Dans notre approche, nous avons utilisé la liste des ressources externes à l'application réifiées par le *middleware* RoSe. Si cela était suffisant pour les cas d'usage que nous avons traité, il est clair que ce contexte deviendra trop limité dans des cas plus complexes. Pour l'augmenter, trois axes peuvent être envisagés :

- l'augmentation de la quantité d'informations disponibles, en incluant dans ce contexte des informations relatives à la plate-forme d'exécution, aux besoins utilisateur, etc.
- la structuration de cette connaissance, en établissant des relations hiérarchiques en termes d'importance et de dépendance entre ses éléments ;
- la modularisation de ce contexte au sein d'un canevas, pour que celui-ci soit facilement extensible et réutilisable.

Bibliographie

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. *SIGSOFT Software Engineering Notes*, 18(5):9–20, December 1993. *2 citations pages 25 et 82*

[AAG95] Gregory D. Abowd, Robert Allen, and David Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 4(4):319–364, October 1995. *cité page 82*

[Abd00] Aynur Abdurazik. Suitability of the UML as an Architecture Description Language with Applications to Testing. Technical report, George Mason University, 2000. *cité page 23*

[ACF+07] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, 2007. *cité page 21*

[ADG98] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In Egidio Astesiano, editor, *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer Berlin Heidelberg, 1998. *cité page 23*

[AEW03] S. Aiber, O. Etzion, and S. Wasserkrug. The utilization of AI techniques in the autonomic monitoring and optimization of business objectives. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-03), Workshop on AI and Autonomic Computing*, August 2003. *cité page 68*

[AFM05] Marco Aiello, Ganna Frankova, and Daniela Malfatti. What's in an Agreement? An Analysis and an Extension of WS-Agreement. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC 2005)*, volume 3826 of

Lecture Notes in Computer Science, pages 424–436. Springer Berlin Heidelberg, 2005. *cité page 28*

[All09] OSGi Alliance. OSGi – the dynamic module system for Java, 2009. *cité page 27*

[All12] ZigBee Alliance. ZigBee specification, 2012. *cité page 4*

[AMS07] Timo Asikainen, Tomi Männistö, and Timo Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Advanced Engineering Informatics*, 21(1):23–40, January 2007. *cité page 43*

[Ars04] Ali Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 2004. *cité page 26*

[ASW+99] Ken Arnold, Robert Scheifler, Jim Waldo, Bryan O'Sullivan, and Ann Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. *cité page 27*

[BAGS02] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002. *cité page 68*

[BBC+03] David F. Bantz, Chatschik Bisdikian, David Challener, John P. Karidis, Steve Mastrianni, Ajay Mohindra, Dennis G. Shea, and Michael Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165–176, 2003. *cité page 70*

[BBFS08] Nelly Bencomo, Gordon S. Blair, Carlos Flores, and Peter Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *Proceedings of the 2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2008)*, 2008. *cité page 51*

[BC13] Jan Bosch and Rafael Capilla. Variability Implementation. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, Lecture Notes in Computer Science, pages 75–86. Springer Berlin Heidelberg, 2013. *cité page 24*

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A Survey of Self-management in Dynamic Software Architecture Specifications. In *Proceedings of the ACM SIGSOFT 2004 Workshop On Self-Managing Systems (WOSS'04)*, pages 28–33, New York, NY, USA, 2004. ACM. *cité page 79*

[BCK12] Len Bass, Paul C. Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 2012. *2 citations pages 23 et 25*

[Bed02] Thomas Bednasch. Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung. 2002. *cité page 40*

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *Proceedings of the 16th IEEE/ACM International Conference on the Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE/ACM, 2001. *cité page 23*

[BGF+08] Nelly Bencomo, Paul Grace, Carlos Flores, Danny Hughes, and Gordon S. Blair. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 811–814, New York, NY, USA, 2008. ACM. *2 citations pages 5 et 51*

[BHSdA12] Nelly Bencomo, Svein Hallsteinsen, and Eduardo Santana de Almeida. A View of the Dynamic Software Product Line Landscape. *Computer*, 45(10):36–41, 2012. *cité page 47*

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In Ron Morrison and Flavio Oquendo, editors, *Software Architecture*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2005. *cité page 86*

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999. *cité page 28*

[Bou08] Johann Bourcier. *Auto-Home : une plate-forme pour la gestion autonomique d'applications pervasives*. Thèse, Université Joseph-Fourier – Grenoble I, 2008. *cité page 77*

[Bro91] Rodney A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1):139–159, 1991. *cité page 68*

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 years Later: a Literature Review. *Information Systems*, 35(6):615–636, 2010. *cité page 50*

[BSTRC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, pages 129–134, 2007. *cité page 50*

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Reasoning on Feature Models. In Oscar Pastor and João Falcão e Cunha, editors, *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE 2005)*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer Berlin Heidelberg, 2005. *2 citations pages 43 et 49*

[CB11] Rafael Capilla and Jan Bosch. The Promise and Challenge of Runtime Variability. *Computer*, 44(12):93–95, 2011. *cité page 47*

[CBG+04] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. 2004. *2 citations pages 85 et 86*

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002. *cité page 39*

[CDK+02] Franscisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web services Web: An Introduction to SOAP, WSDL, and UDDI. *Internet Computing, IEEE*, 6(2):86–93, 2002. *cité page 27*

[CDM91] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Distributed optimization by ant colonies. In *Proceedings of the 1st European conference on artificial life*, volume 142, pages 134–142, 1991. *cité page 66*

[Cer04] Humberto Cervantes. *Vers un modèle à composants orienté services pour supporter la disponibilité dynamique*. Thèse, Université Joseph-Fourier – Grenoble I, 2004. *3 citations pages 6, 17, et 33*

[CFP08] Carlos Cetina, Joan Fons, and Vincente Pelechano. Applying Software Product Lines to Build Autonomic Pervasive Systems. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 117–126, 2008. *2 citations pages 48 et 49*

[CGFP09a] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. *Computer*, 42(10):37–43, 2009. *cité page 48*

[CGFP09b] Carlos Cetina, Pau Giner, Joan Fons, and Vincente Pelechano. Using Feature Models for Developing Self-Configuring Smart Homes. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009)*, pages 179–188, 2009. *cité page 49*

[CGR+12] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2012)*, pages 173–182, New York, NY, USA, 2012. ACM. *cité page 42*

[CGS+02] Shang-Wen Cheng, David Garlan, Bradley Schmerl, João Pedro Sousa, Bridget Spitznagel, and Peter Steenkiste. Using Architectural Style as a Basis for System Self-repair. In *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 45–59, 2002. *2 citations pages 82 et 84*

[CGS09] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In *Proceedings of the ICSE 2009 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009)*, pages 132–141, 2009. *cité page 83*

[CH03] Humberto Cervantes and Richard S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering (CBSE)*, 2003. *cité page 34*

[CH04] Humberto Cervantes and Richard S. Hall. Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society. *3 citations pages 6, 33, et 34*

[CHE04] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration Using Feature Models. In Robert L. Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin Heidelberg, 2004. *2 citations pages 38 et 39*

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. *2 citations pages 39 et 43*

[Che08] Shang-Wen Cheng. *Rainbow: Cost-effective, Software Architecture-based Self-adaptation*. Thèse, School of Computer Science Carnegie Mellon University, 2008. *5 citations pages 80, 81, 83, 84, et 202*

[CHG+04] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. An architecture for coordinating multiple self-management systems. In *4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*, pages 243–252. IEEE Computer Society, 2004. *2 citations pages 25 et 81*

[Cho09] Stéphanie Chollet. *Orchestration de services hétérogènes et sécurisés*. Thèse, Université Joseph-Fourier – Grenoble I, December 2009. *2 citations pages 26 et 27*

[CHZ+09] Carlos Cetina, Øystein Haugen, Xiaorui Zhang, Franck Fleurey, and Vicente Pelechano. Strategies for variability transformation at run-time. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 61–70, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. *cité page 49*

[CK05] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA, 2005. ACM, ACM. *cité page 39*

[CKB13] Rafael Capilla, Kyo-Chul Kang, and Jan Bosch. *Systems and Software Variability Management*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013. *cité page 14*

[Cle02] Paul C. Clements. On the Importance of Product Line Scope. In *Revised Papers from the 4th International Workshop on Software Product-Family Engineering (PFE'01)*, Lecture Notes in Computer Science, pages 70–78, London, UK, 2002. Springer-Verlag. *cité page 38*

[CLM+12] Stéphanie Chollet, Vincent Lestideau, Yoann Maurel, Etienne Gandrille, Philippe Lalanda, and Olivier Raynaud. Practical Use of Formal Concept Analysis in Service-Oriented Computing. In Florent Domenach, Dmitry I. Ignatov, and Jonas Poelmans, editors, *Proceedings of the 10th International Conference on Formal Concept Analysis (ICFCA 2012)*, volume 7278 of *Lecture Notes in Computer Science (LNCS)*, pages 61–76, Leuven, Belgium, May 2012. Springer. *cité page 108*

[CN02] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002. *2 citations pages 7 et 37*

[CPGS09] Shang-Wen Cheng, Vahe V. Poladian, David Garlan, and Bradley Schmerl. Improving Architecture-Based Self-Adaptation through Resource Prediction. In Betty H.C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 71–88. Springer Berlin Heidelberg, 2009. *cité page 83*

[CPRS04] Vaclav Cechticky, Alessandro Pasetti, Ondřej Rohlik, and Walter Schaufelberger. XML-Based Feature Modelling. In Jan Bosch and Charles Krueger, editors, *Software Reuse: Methods, Techniques, and Tools*, volume 3107 of *Lecture Notes in Computer Science*, pages 101–114. Springer Berlin Heidelberg, 2004. *cité page 39*

[CSVC11] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel R.V. Chaudron. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering*, 37(5):593–615, september–october 2011. *2 citations pages ix et 21*

[CTP10] Arnaud Cogoluegnes, Thierry Templier, and Andy Piper. *Spring Dynamic Modules in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. *cité page 34*

[DAH+07] Eric Matthew Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. ArchStudio 4: An Architecture-Based Meta-Modeling Environment. In *Companion to the Proceedings of the 29th International Conference on Software Engineering (ICSE COMPANION'07)*, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society. *cité page 91*

[Das07] Eric Matthew Dashofy. *Supporting Stakeholder-driven, Multi-view Software Architecture Modeling*. Thèse, University of California, Irvine, 2007. *cité page 92*

[Deb14] Bassem Debbabi. *Cube: a decentralised architecture-based framework for software self-management*. Thèse, Université de Grenoble, 2014. *2 citations pages 175 et 205*

[DFT92] John Comstock Doyle, Bruce A. Francis, and Allen Tannenbaum. *Feedback control theory*, volume 1. Macmillan Publishing Company New York, 1992. *cité page 67*

[DL12] Ana Dragomir and Horst Lichter. Model-Based Software Architecture Evolution and Evaluation. In *Proceedings of the 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*, volume 1, pages 697–700, 2012. *cité page 23*

[Dor02] Andy Dornan. *The Essential Guide to Wireless Communications Applications*. Prentice Hall, 2nd edition edition, 2002. *cité page 4*

[DvdHT02] Eric Matthew Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the 1st ACM SIGSOFT Workshop On Self-healing Systems (WOSS'02)*, pages 21–26, New York, NY, USA, 2002. ACM. *cité page 92*

[DvdHT05] Eric Matthew Dashofy, André van der Hoek, and Richard N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 14(2):199–245, April 2005. *cité page 92*

[DWH03] Tom De Wolf and Tom Holvoet. Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control. In *Proceedings of the IEEE International Conference on Industrial Informatics (INDIN'2003)*, pages 470–479. IEEE, 2003. *cité page 67*

[DWH06] Tom De Wolf and Tom Holvoet. *Autonomic computing: concepts, infrastructure, and applications*, chapter A Taxonomy for Self-* Properties in Decentralised Autonomic Computing, pages 101–120. CRC Press, Taylor and Francis Group, 2006. *cité page 72*

[EAA+04] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. *Patterns: service-oriented architecture and web services*. IBM, 2004. www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf. *2 citations pages ix et 31*

[eAMO11] Fazal e Amin, Ahmad Kamil Mahmood, and Alan Oxley. An analysis of object oriented variability implementation mechanisms. *SIGSOFT Software Engineering Notes*, 36(1):1–4, January 2011. *cité page 16*

[ECL14] Clément Escoffier, Stéphanie Chollet, and Philippe Lalanda. Lessons learned in building pervasive platforms. In *11th IEEE Consumer Communications and Networking Conference (CCNC 2014)*, pages 7–12, 2014. *cité page 175*

[EEM13] Naeem Esfahani, Ahmed Elkhodary, and Sam Malek. A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems. *IEEE Transactions on Software Engineering*, 39(11):1467–1493, November 2013. *cité page 90*

[EHL07] Clément Escoffier, Richard S. Hall, and Philippe Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *Proceedings of the IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481. IEEE Computer Society, 2007. *3 citations pages ix, 14, et 34*

[EM08] George Edwards and Nenad Medvidovic. A Methodology and Framework for Creating Domain-Specific Development Infrastructures. In *prodeedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 168–177. IEEE, 2008. *cité page 90*

[EMM07] George Edwards, Sam Malek, and Nenad Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architectures. In Matthew B. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422 of *Lecture Notes in Computer Science*, pages 125–139. Springer Berlin Heidelberg, 2007. *cité page 90*

[Esc08] Clément Escoffier. *iPOJO : Un modèle à composant à service flexible pour les systèmes dynamiques*. Thèse, Université Joseph-Fourier – Grenoble I, December 2008. *3 citations pages 29, 32, et 34*

[Fab76] Robert S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE 1976)*, pages 470–476. IEEE Computer Society Press, IEEE Computer Society, 1976. *cité page 22*

[Fav04] Jean-Marie Favre. Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of the Fidus Papyrus and of the Solarus. In *Postproceedings of Dagsthul Seminar on Model Driven Reverse Engineering*, 2004. *cité page 101*

[FCBG07] Carlos Flores-Cortes, Gordon S. Blair, and Paul Grace. An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. *IEEE Distributed Systems Online*, 8(7):1, 2007. *cité page 51*

[FDB+09] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and Validating Dynamic Adaptation. In Michel R.V. Chaudron, editor, *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 97–108. Springer Berlin Heidelberg, 2009. *cité page 53*

[FDB+12] François Fouquet, Erwan Daubert, Olivier Barais, Noël Plouzeau, Johann Bourcier, Jean-Emile Dartois, and Arnaud Blouin. Kevoree: une approche model@ runtime pour les systèmes ubiquitaires. In *Proceedings of the 8èmes journées francophones Mobilité et Ubiquité (Ubimob 2012)*, 2012. *cité page 5*

[FECA04] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit. *Aspect-oriented Software Development*. Addison-Wesley Professional, first edition, 2004. *cité page 48*

[FHS+06] Jacqueline Floch, Svein Hallsteinsen, Erlend Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. Using architecture models for runtime adaptability. *Software, IEEE*, 23(2):62–70, 2006. *2 citations pages 5 et 49*

[FHS08] Luís Fraga, Svein Hallsteinsen, and Ulrich Scholz. "InstantSocial"–Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware. *Electronic Communications of the European Association of Software Science and Technology (EASST)*, 11:1–7, 2008. *cité page 54*

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of the 29th International Conference on Software Engineering (ICSE 2007) – Future Of Software Engineering track (FOSE)*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. *2 citations pages 7 et 203*

[Gar13] David Garlan. A 10-year Perspective on Software Engineering Self-adaptive Systems. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)*, pages 2–2, Piscataway, NJ, USA, 2013. IEEE Press. *4 citations pages 13, 25, 56, et 205*

[Gat98] Erann Gat. On three-layer architectures. *Artificial intelligence and mobile robots*, pages 195–210, 1998. *cité page 87*

[GAWM11] Matthias Galster, Paris Avgeriou, Danny Weyns, and Tomi Männistö. Variability in software architecture: current practice and challenges. *SIGSOFT Software Engineering Notes*, 36(5):30–32, September 2011. *3 citations pages 24, 25, et 56*

[GBE+09] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjorven, Svein Hallsteinsen, Geir Horn, Mohammad Ullah Khan, Alessandro Mamelli, George A. Papadopoulos, Nearchos Paspallis, Roland Reichle, and Erlend Stav. A comprehensive solution for application-level adaptation. *Software: Practice and Experience*, 39(4):385–422, 2009. *cité page 49*

[GBJC07] Antônio Tadeu Azevedo Gomes, Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Architecting Dynamic Reconfiguration in Dependable Systems. In Rogério Lemos, Cristina Gacek, and Alexander Romanovsky, editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*, pages 237–261. Springer Berlin Heidelberg, 2007. *2 citations pages x et 85*

[GC03] Alan G. Ganek and Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003. *cité page 63*

[GCBP05] Paul Grace, Geoff Coulson, Gordon S. Blair, and Barry Porter. Deep Middleware for the Divergent Grid. In Gustavo Alonso, editor, *Middleware 2005*, volume 3790 of *Lecture Notes in Computer Science*, pages 334–353. Springer Berlin Heidelberg, 2005. *cité page 51*

[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, 2004. *3 citations pages 5, 79, et 93*

[GDL+04] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems (TOCS)*, 22(4):421–486, November 2004. *3 citations pages 3, 4, et 5*

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, pages 76–85, 1998. *cité page 39*

[GG12] Issac Noé García Garza. *Modèles de conception et d'exécution pour la médiation et l'intégration de services*. Thèse, Université de Grenoble, 2012. *2 citations pages 158 et 170*

[GGMD+11] Issac Noé García Garza, Denis Morand, Bassem Debbabi, Philippe Lalanda, and Pierre Bourret. A Reflective Framework for Mediation Applications. In *Proceedings of the 12th International Middleware Conference*, pages 22–28. ACM, December 2011. *3 citations pages xiii, 162, et 197*

[GH04] Hassan Gomaa and Mohamed Hussein. Dynamic Software Reconfiguration in Software Product Families. In Frank J. Linden, editor, *Software Product-Family Engineering*, volume 3014 of *Lecture Notes in Computer Science*, pages 435–444. Springer Berlin Heidelberg, 2004. *cité page 46*

[GH11] Hassan Gomaa and Koji Hashimoto. Dynamic Software Adaptation for Service-oriented Product Lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC 2011), Volume 2*, pages 35:1–35:8, New York, NY, USA, 2011. ACM. *cité page 54*

[GH12] Hassan Gomaa and Koji Hashimoto. Dynamic self-adaptation for distributed service-oriented transactions. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012)*, pages 11–20, 2012. *2 citations pages 14 et 54*

[GHJV95] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*, volume 49. Addison-Wesley, 1995. *2 citations pages 18 et 22*

[GHL13a] Etienne Gandrille, Catherine Hamon, and Philippe Lalanda. Design and runtime architectures to support autonomic management. In Anca Daniela Ionita, Grace A. Lewis, and Marin Litoiu, editors, *IEEE 7th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, pages 75–84, Eindhoven, The Netherlands, September 2013. IEEE. *cité page 100*

[GHL13b] Etienne Gandrille, Catherine Hamon, and Philippe Lalanda. Linking Reference and Runtime Architectures in Autonomic Systems. In *STO-MP-IST-115 - Architecture Definition and Evaluation*, pages 17–1 – 17–7. NATO Science and Technology Organization, May 2013. *cité page 100*

[GM03] Dimitra Giannakopoulou and Jeff Magee. Fluent Model Checking for Event-based Systems. In *Proceedings of the 9th European Software Engineering Conference (ESEC) Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 257–266, New York, NY, USA, 2003. ACM. *cité page 88*

[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. *cité page 85*

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004. *cité page 88*

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004. *2 citations pages 54 et 55*

[GP08] Dimitrios Georgakopoulos and Michael P. Papazoglou. *Service-Oriented Computing*. The MIT Press, 2008. *cité page 31*

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. *cité page 17*

[Gra59] Pierre-Paul Grassé. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie : Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, 1959. *cité page 66*

[GS04] David Garlan and Bradley Schmerl. Using Architectural Models at Runtime: Research Challenges. In Flavio Oquendo, Brian C. Warboys, and Ron Morrison, editors, *Software Architecture*, volume 3047 of *Lecture Notes in Computer Science*, pages 200–205. Springer Berlin Heidelberg, 2004. *cité page 23*

[GSC01] David Garlan, Bradley Schmerl, and Jichuan Chang. Using Gauges for Architecture-Based Monitoring and Adaptation. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, 12-14 December 2001. *cité page 80*

[GSC09] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. Software Architecture-Based Self-Adaptation. In Yan Zhang, Laurence Tianruo Yang, and Mieso K. Denko, editors, *Autonomic Computing and Networking*, Lecture Notes in Computer Science, pages 31–55. Springer, 2009. *8 citations pages x, 24, 74, 77, 79, 80, 83, et 93*

[GSRU07] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems – survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007. *cité page 66*

[GT00] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *Recent Advances in AI Planning*, pages 1–20. Springer, 2000. *cité page 88*

[GT08] John C. Georgas and Richard N. Taylor. Policy-based Self-adaptive Architectures: A Feasibility Study in the Robotics Domain. In *Proceedings of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, pages 105–112, New York, NY, USA, 2008. ACM. *cité page 92*

[GTTGPn+11] Antonio González-Torres, Roberto Therón, Francisco J. Garcóa-Peñalvo, Michel Wermelinger, and Yijun Yu. Maleku: An evolutionary visual software analysis tool for providing insights into software evolution. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pages 594–597, 2011. *cité page 23*

[Guo03] Haipeng Guo. A bayesian approach for automatic algorithm selection. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence (IJCAI-03), Workshop on AI and Autonomic Computing*, August 2003. *cité page 68*

[GvdHT05] John C. Georgas, André van der Hoek, and Richard N. Taylor. Architectural Runtime Configuration Management in Support of Dependable Self-adaptive Software. In *Proceedings of the 2005 Workshop on Architecting Dependable Systems (WADS'05)*, pages 1–6, New York, NY, USA, 2005. ACM. *cité page 91*

[GvdHT09] John C. Georgas, André van der Hoek, and Richard N. Taylor. Using Architectural Models to Manage and Visualize Runtime Adaptation. *Computer*, 42(10):52–60, Oct 2009. *2 citations pages x et 91*

[Hay03] Brian Hayes. The post-OOP paradigm. *American Scientist*, 91(2):106–110, 2003. *cité page 17*

[HC01] George T. Heineman and William T. Councill. *Component-based software engineering: putting the pieces together*, volume 17. Addison-Wesley Reading, 2001. *cité page 20*

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. *cité page 13*

[HHPS08] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008. *cité page 46*

[HHPS13] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, Lecture Notes in Computer Science, pages 253–260. Springer Berlin Heidelberg, 2013. *2 citations pages 46 et 47*

[Hor01] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology, October 2001. *2 citations pages 61 et 63*

[HPS12] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building Dynamic Software Product Lines. *Computer*, 45(10):22–26, 2012. *cité page 46*

[HS12] Mike Hinchey and Roy Sterritt. 99% (Biological) Inspiration.... In Mike Hinchey and Lorcan Coyle, editors, *Conquering Complexity*, pages 177–190. Springer London, 2012. *cité page 66*

[HSMK09] William Heaven, Daniel Sykes, Jeff Magee, and Jeff Kramer. A Case Study in Goal-Driven Architectural Adaptation. In Betty H.C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 109–127. Springer Berlin Heidelberg, 2009. *cité page 88*

[HSSF06] Svein Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Proceedings of the 10th International Software Product Line Conference (SPLC 2006)*, pages 10–150, 2006. *cité page 49*

[HWS+09] Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Proceedings of the 3rd SPLC International Workshop on Dynamic Software Product Lines (DSPL 2009)*, pages 18–27, 2009. *cité page 47*

[IBM06] IBM. An architectural blueprint for autonomic computing. *IBM White Paper*, 2006. *10 citations pages ix, 63, 64, 65, 66, 73, 76, 77, 94, et 170*

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua – an Extensible Extension Language. *Software Practice & Experience*, 26(6):635–652, June 1996. *cité page 86*

[Jac02] Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 11(2):256–290, April 2002. *cité page 88*

[JBCG05] Ackbar Joolia, Thaís Vasconcelos Batista, Geoff Coulson, and Antônio Tadeu Azevedo Gomes. Mapping ADL Specifications to an Efficient and Reconfigurable Runtime Component Platform. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*, pages 131–140. IEEE Computer Society, 2005. *cité page 86*

[JFR06] JFR. Z-Wave Protocol Overview (version 2). Technical report, Zensys A/S, 2006. *cité page 4*

[JHB+10] Shanshan Jiang, Svein Hallsteinsen, Paolo Barone, Alessandro Mamelli, Stephan Mehlhase, and Ulrich Scholz. Hosting and Using Services with QoS Guarantee in Self-adaptive Service Systems. In Frank Eliassen and Rüdiger Kapitza, editors, *Distributed Applications and Interoperable Systems*, volume 6115 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin Heidelberg, 2010. *cité page 54*

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. *6 citations pages 62, 63, 68, 70, 73, et 100*

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990. *4 citations pages ix, 38, 39, et 103*

[KD07] Jeffrey O. Kephart and Rajarshi Das. Achieving Self-Management via Utility Functions. *Internet Computing, IEEE*, 11(1):40–48, 2007. *cité page 68*

[Kep05] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 15–22. ACM, 2005. *cité page 74*

[KKP+09] Minseong Kim, Suntae Kim, Sooyong Park, Mun-Taek Choi, Munsang Kim, and Hassan Gomaa. Service robot for the elderly. *Robotics Automation Magazine, IEEE*, 16(1):34–45, 2009. *cité page 54*

[KLR09] Gerald Kotonya, Jaejoon Lee, and Daniel Robinson. A consumer-centred approach for service-oriented product line development. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture, European Conference on Software Architecture (WICSA/ECSA 2009)*, pages 211–220. IEEE, 2009. *cité page 55*

[KM06] Jeff Kramer and Jeff Magee. *Concurrency: State Models and Java Programming*. John Wiley and Sons, 2nd edition, 2006. *cité page 88*

[KM09] Jeff Kramer and Jeff Magee. A Rigorous Architectural Approach to Adaptive Software Engineering. *Journal of Computer Science and Technology*, 24(2):183–188, 2009. *2 citations pages x et 87*

[KW04] Jeffrey O. Kephart and William E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Proceedings of the 5th IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2004)*, pages 3–12. IEEE Computer Society, 2004. *cité page 75*

[LCGH14] Philippe Lalanda, Stéphanie Chollet, Etienne Gandrille, and Catherine Hamon. Maintaining Traceability Links between Design and Runtime Architectures to support Autonomic Management. In Frank Golatowski, editor, *Proceedings of the 8th International Workshop on Service-Oriented Cyber-Physical Systems in Converging Networked Environments (SOCNE)*, pages 19–24, Barcelona, Spain, 2014. *cité page 100*

[LDM13] Philippe Lalanda, Ada Diaconescu, and Julie A. McCann. *Autonomic Computing – Principles, Design and Implementation*. Undergraduate Topics in Computer Science. Springer, May 2013. *4 citations pages ix, 13, 22, et 77*

[LK13] Jaejoon Lee and Gerald Kotonya. Service-Oriented Product Lines. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, Lecture Notes in Computer Science, pages 279–285. Springer Berlin Heidelberg, 2013. *cité page 55*

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In Cristina Gacek, editor, *Software Reuse: Methods, Techniques, and Tools*, volume 2319 of *Lecture Notes in Computer Science*, pages 62–77. Springer Berlin Heidelberg, 2002. *cité page 39*

[LKR12] Jaejoon Lee, Gerald Kotonya, and Daniel Robinson. Engineering Service-Based Dynamic Software Product Lines. *Computer*, 45(10):49–55, 2012. *cité page 55*

[LM08] Mikael Lindvall and Dirk Muthig. Bridging the Software Architecture Gap. *Computer*, 41(6):98–101, 2008. *cité page 23*

[LMN08] Jaejoon Lee, Dirk Muthig, and Matthias Naab. An Approach for Developing Service Oriented Product Lines. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 275–284, 2008. *cité page 55*

[LMV+07] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 498–513. Springer Berlin Heidelberg, 2007. *cité page 52*

[LSR07] Frank Linden, Klaus Schmid, and Eelco Rommes. Telvent. In *Software Product Lines in Action*, Lecture Notes in Computer Science, pages 265–274. Springer Berlin Heidelberg, 2007. *cité page 50*

[Luc96] David C. Luckham. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events. In *Proceedings of the DIMACS Partial Order Methods Workshop*. Princeton University, 1996. *cité page 23*

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In Norman K. Meyrowitz, editor, *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA'87)*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM. *cité page 162*

[Mai02] Evaristus Mainsah. Autonomic computing: the next era of computing. *Electronics Communication Engineering Journal*, 14(1):2–3, 2002. *cité page 61*

[Man02] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In Gary J. Chastek, editor, *Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer Berlin Heidelberg, 2002. *cité page 43*

[Mat01] Friedemann Mattern. The Vision and Technical Foundations of Ubiquitous Computing. *Upgrade*, 2(5):2–6, October 2001. *2 citations pages 3 et 4*

[Mau10] Yoann Maurel. *CEYLAN : Un canevas pour la création de gestionnaires autonomiques extensibles et dynamiques*. Thèse, Université de Grenoble, 2010. *cité page 77*

[MBJ08] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *Proceedings of the 2nd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2008)*, Essen, Germany, Allemagne, 2008. *cité page 52*

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009. *3 citations pages 7, 46, et 52*

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems Using Models and Aspects. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society. *cité page 52*

[MCL+12] Yoann Maurel, Stéphanie Chollet, Vincent Lestideau, Jonathan Bardin, Philippe Lalanda, and André Bottaro. fANFARE: Autonomic Framework for Service-based Pervasive Environment. In *International Conference on Service Computing (SCC 2012)*, pages 65–72, Honolulu, HI, United States, June 2012. IEEE. Research Session 3 - Application Modeling and Management. *cité page 108*

[McV09] Andrew McVeigh. *A Rigorous, Architectural Approach to Extensible Applications*. Thèse, Imperial College London, Department of Computing, August 2009. *cité page 88*

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In Wilhelm Schäfer and Pere Botella, editors, *Proceedings of the 5th European Software Engineering Conference (ESEC'95)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin Heidelberg, 1995. *cité page 87*

[MEG+10] Daniel A. Menascé, John M. Ewing, Hassan Gomaa, Sam Malex, and João P. Sousa. A Framework for Utility-based Service Oriented Design in SASSY. In *Proceedings of the 1st Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 27–36, New York, NY, USA, 2010. ACM. *cité page 54*

[MEM+09] Sam Malek, Naeem Esfahani, Daniel A. Menasce, João P. Sousa, and Hassan Gomaa. Self-Architecting Software SYstems (SASSY) from QoS-annotated activity models. In *Proceedings of the 1st ICSE International Workshop on Principles of Engineering Service-Oriented and Cloud Systems (PESOS)*, pages 62–69, 2009. *cité page 54*

[Mey92] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. *cité page 28*

[MFB+08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon S. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 782–796. Springer Berlin Heidelberg, 2008. *cité page 53*

[MGMS11] Daniel A. Menasce, Hassan Gomaa, Sam Malek, and Jo ao P. Sousa. SASSY: A Framework for Self-Architecting Service-Oriented Systems. *Software, IEEE*, 28(6):78–85, 2011. *cité page 54*

[MH04] Julie A. McCann and Markus C. Huebscher. Evaluation Issues in Autonomic Computing. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *Grid and Cooperative Computing – GCC 2004 Workshops*, volume 3252 of *Lecture Notes in Computer Science*, pages 597–608. Springer Berlin Heidelberg, 2004. *cité page 67*

[Mil87] George W. (Bill) Miller. Service Level Agreements: "Good Fences Make Good Neighbors". In *Proceedings of the 13th International Computer Measurement Group Conference (CMG)*, pages 553–558. Computer Measurement Group, 1987. *cité page 28*

[Mil05] Brent Miller. The autonomic computing edge: Can you CHOP up autonomic computing, 2005. http://www.ibm.com/developerworks/library/ac-edge4/. *cité page 70*

[MK96] Jeff Magee and Jeff Kramer. Dynamic Structure in Software Architectures. In *Proceedings of the 4th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 3–14, New York, NY, USA, 1996. ACM. *cité page 87*

[MKM06] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using Resemblance to Support Component Reuse and Evolution. In *Proceedings of the 2006 Conference on Specification and Verification of Component-based Systems (SAVCBS '06) : 5th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 49–56, New York, NY, USA, 2006. ACM. *cité page 88*

[MKM11] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Evolve: Tool Support for Architecture Evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1040–1042, New York, NY, USA, 2011. ACM. *cité page 88*

[MLM+06] C. Matthew MacKenzie, Ken Laskey, Francis McCabe, Peter F. Brown, Rebekah Metz, and Booz Allen Hamilton. Reference model for service oriented architecture 1.0. *OASIS Standard*, 12, 2006. *cité page 26*

[MMMR12] Sam Malek, Nenad Medvidovic, and Marija Mikic-Rakic. An Extensible Framework for Improving a Distributed Software System's Deployment Architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, Jan 2012. *4 citations pages x, 88, 89, et 90*

[MMR10] Tom Mens, Jeff Magee, and Bernhard Rumpe. Evolving software architecture descriptions of critical systems. *Computer*, 43:42–48, 2010. *cité page 88*

[MMRM05] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, March 2005. *2 citations pages 89 et 90*

[MN97] Gail C. Murphy and David Notkin. Reengineering with reflexion models: a case study. *Computer*, 30(8):29–36, 1997. *cité page 23*

[MOKW06] Hausi A. Müller, Liam O'Brien, Mark Klein, and Bill Wood. Autonomic computing. Technical report, DTIC Document, 2006. *2 citations pages 66 et 69*

[Mon00] Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical report, Carnegie Mellon University, USA, September 2000. *2 citations pages 85 et 86*

[Mor13] Denis Morand. *Cilia : un framework pour le développement d'applications de médiation autonomiques*. Thèse, Université de Grenoble, 2013. *5 citations pages 74, 101, 158, 162, et 197*

[MP06] Javier Muñoz and Vicente Pelechano. Applying Software Factories to Pervasive Systems: A Platform Specific Framework. In *Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS 2006)*, pages 337–342, 2006. *cité page 49*

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000. *cité page 79*

[MT10] Nenad Medvidovic and Richard N. Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010) – Volume 2*, pages 471–472, New York, NY, USA, 2010. ACM. *cité page 22*

[NBT+11] Norbert Noury, Marc Berenguer, Henri Teyssier, Marie-Jeanne Bouzid, and Michel Giordani. Building an Index of Activity of Inhabitants From Their Activity on the Residential Electrical Power Line. *IEEE Transactions on Information Technology in Biomedicine*, 15(5):758–766, Sept 2011. *cité page 178*

[ND95] Oscar Nierstrasz and Laurent Dami. Component-oriented software technology. *Object-Oriented Software Composition*, 1:3–28, 1995. *cité page 20*

[NDBJ08] Grégory Nain, Erwan Daubert, Olivier Barais, and Jean-Marc Jézéquel. Using MDE to Build a Schizophrenic Middleware for Home/Building Automation. In Petri Mähönen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 49–61. Springer Berlin Heidelberg, 2008. *cité page 53*

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 72–83, New York, NY, USA, 2006. ACM. *cité page 16*

[Nor02] Linda M. Northrop. SEI's software product line tenets. *Software, IEEE*, 19(4):32–40, 2002. *cité page 43*

[NQB+09] Norbert Noury, Kim Anh Quach, Marc Berenguer, Henri Teyssier, Marie-Jeanne Bouzid, Laurent Goldstein, and Michel Giordani. Remote follow up of health through the monitoring of electrical activities on the residential power line - preliminary results of an experimentation. In *11th International Conference on e-Health Networking, Applications and Services (Healthcom 2009)*, pages 9–13, Dec 2009. *cité page 178*

[NQB+10] Norbert Noury, Kim Anh Quach, Marc Berenguer, Henri Teyssier, Marie-Jeanne. Bouzid, Laurent Goldstein, and Michel Giordani. Ubiquitous but non invasive evaluation of the activity of a person from a unique index built on the electrical activities on the residential power line. In *12th IEEE International Conference on e-Health Networking Applications and Services (Healthcom 2010)*, pages 1–6, July 2010. *cité page 178*

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimhigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, 1999. *5 citations pages 5, 24, 56, 79, et 93*

[OMG02] OMG. Meta-Object Facility (MOF™) Specification, April 2002. version 1.4. *cité page 101*

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society. *2 citations pages 79 et 93*

[Pap03] Mike P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003)*, pages 3–12, 2003. *3 citations pages ix, 26, et 27*

[Par11] Carlos Parra. *Towards Dynamic Software Product Lines: Unifying Design and Runtime Adaptations*. Thèse, Université des Sciences et Technologie de Lille – Lille I, March 2011. *cité page 51*

[PBCD11] Carlos Parra, Xavier Blanc, Anthony Cleve, and Laurence Duchien. Unifying Design and Runtime Software Adaptation Using Aspect Models. *Science of Computer Programming*, 76(12):1247–1260, December 2011. *cité page 51*

[PBD09] Carlos Parra, Xavier Blanc, and Laurence Duchien. Context Awareness for Dynamic Service-oriented Product Lines. In *Proceedings of the 13th International Software Product Line Conference (SPLC 2009)*, pages 131–140, Pittsburgh, PA, USA, 2009. Carnegie Mellon University. *cité page 51*

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. *8 citations pages ix, 37, 40, 41, 42, 43, 44, et 45*

[PC08] Petros Pissias and Geoff Coulson. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *IET Software*, 2(4):348–361, Aug 2008. *cité page 85*

[PH05] Manish Parashar and Salim Hariri. Autonomic Computing: An Overview. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 257–269. Springer Berlin Heidelberg, 2005. *2 citations pages 63 et 66*

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007. *5 citations pages ix, 26, 30, 31, et 32*

[PQD12] Carlos Parra, Clément Quinton, and Laurence Duchien. CAPucine: Context-Aware Service-Oriented Product Line for Mobile Apps. *ERCIM News*, 88:38–39, January 2012. *cité page 51*

[PRK10] Joern Ploennigs, Uwe Ryssel, and Klaus Kabitzsch. Performance analysis of the EnOcean wireless sensor network protocol. In *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–9, 2010. *cité page 4*

[PTD+06] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. Service-Oriented Computing Research Roadmap. *Dagstuhl Seminar Proceedings 05462*, pages 1–29, 2006. *cité page 26*

[RBC05] Christopher Roblee, Vincent Berk, and George Cybenko. Implementing Large-Scale Autonomic Server Monitoring Using Process Query Systems. In *Proceedings of the 2nd International Conference on Autonomic Computing (ICAC'05)*, pages 123–133, 2005. *cité page 74*

[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 Betty H.C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 164–182. Springer Berlin Heidelberg, 2009. *cité page 54*

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending Feature Diagrams With Uml Multiplicities, 2002. *cité page 39*

[RCAM+05] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, and M. Dennis Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005)*, pages 7–16. IEEE Computer Society, 2005. *cité page 5*

[RCS08] Romain Rouvoy, Denis Conan, and Lionel Seinturier. Software Architecture Patterns for a Context-Processing Middleware Framework. *Distributed Systems Online, IEEE*, 9(6):1–1, 2008. *cité page 52*

[REF+08] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. Composing Components and Services Using a Planning-Based Adaptation Middleware. In Cesare Pautasso and Éric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 52–67. Springer Berlin Heidelberg, 2008. *cité page 54*

[RF09] Fabricia Roos-Frantz. A Preliminary Comparison of Formal Properties on Orthogonal Variability Model and Feature Models. In *Proceedings of the 3rd International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2009)*, pages 121–126, 2009. *cité page 42*

[RK76] Howard Raiffa and Ralph L. Keeney. *Decisions with multiple objectives: Preferences and value tradeoffs*. Cambridge University Press, 1976. *cité page 68*

[RNC+95] Stuart Jonathan Russell, Peter Norvig, John F. Canny, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995. *cité page 68*

[Ron09] Daniel Ronzani. The battle of concepts: Ubiquitous Computing, pervasive computing and ambient intelligence in Mass Media. *Ubiquitous Computing and Communication Journal*, 4(2):9–19, 2009. *cité page 2*

[Sat01] Mahadev Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10–17, 2001. *cité page 3*

[SB03a] Roy Sterritt and Dave Bustard. Autonomic Computing – a means of achieving dependability? In *Proceedings of the 10th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2003)*, pages 247–251. IEEE, 2003. *4 citations pages ix, 70, 71, et 72*

[SB03b] Roy Sterritt and Dave Bustard. Towards an autonomic computing environment. In *Proceedings of the 14th International Workshop on Database and Expert Systems Applications*, pages 694–698, 2003. *cité page 71*

[SB11] Zach Shelby and Carsten Bormann. *6LoWPAN: The wireless embedded Internet*, volume 43. John Wiley & Sons, 2011. *cité page 4*

[SD07] Marco Sinnema and Sybren Deelstra. Classifying variability modeling techniques. *Information Software Technologies*, 49(7):717–739, July 2007. *cité page 38*

[Sei03] Ed Seidewitz. What models mean. *Software, IEEE*, 20(5):26–32, 2003. *cité page 23*

[SEMD10] Dale E. Seborg, Thomas F. Edgar, Duncan A. Mellichamp, and Francis J. Doyle. *Process dynamics and control*. John Wiley & Sons, 3rd edition, 2010. *2 citations pages 67 et 82*

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996. *cité page 23*

[SG98] Bridget Spitznagel and David Garlan. Architecture-Based Performance Analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering (SEKE'98)*, June 1998. *cité page 82*

[SG02] Bradley Schmerl and David Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th international conference on Software Engineering and Knowledge Engineering (SEKE'2002)*, pages 241–248, New York, NY, USA, 2002. ACM. *cité page 99*

[SHB10] Gregor Schiele, Marcus Handte, and Christian Becker. Pervasive Computing Middleware. In Hideyuki Nakashima, Hamid Aghajan, and Juan-Carlos Augusto, editors, *Handbook of Ambient Intelligence and Smart Environments*, Lecture Notes in Computer Science, pages 201–227. Springer US, 2010. *cité page 5*

[SHMK07] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Plan-directed Architectural Change for Autonomous Systems. In *Proceedings of the sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 15–21, New York, NY, USA, 2007. ACM. *2 citations pages 87 et 88*

[SHMK08] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. From Goals to Components: A Combined Approach to Self-management. In *Proceedings of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008)*, pages 1–8, New York, NY, USA, 2008. ACM. *cité page 87*

[SHMK10] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting Non-functional Preferences in Architectural Adaptation for Self-managed Systems. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC 2010)*, pages 431–438, New York, NY, USA, 2010. ACM. *cité page 88*

[Sim11] Eric Simon. *SAM : un environnement d'exécution pour les applications à services dynamiques et hétérogènes*. Thèse, Université de Grenoble, March 2011. *2 citations pages 26 et 36*

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing (SCC 2009)*, pages 268–275. IEEE Computer Society, 2009. *cité page 34*

[SS98] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems: design and evaluation*, volume 3. AK Peters Massachusetts, 1998. *cité page 70*

[SS05] Junichi Suzuki and Tatsuya Suda. A middleware platform for a biologically inspired network architecture supporting autonomous and adaptive applications. *Selected Areas in Communications, IEEE Journal on*, 23(2):249–260, 2005. *cité page 66*

[SSB05] Roy Sterritt, Barry Smyth, and Martin Bradley. PACT personal autonomic computing tools. In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2005)*, pages 519–527. IEEE, 2005. *cité page 74*

[Ste02] Roy Sterritt. Towards autonomic computing: effective event management. In *Proceedings of the 27th Annual NASA Goddard/IEEE Workshop on Software Engineering*, pages 40–47, 2002. *cité page 71*

[Ste05] Roy Sterritt. Autonomic computing. *Innovations in Systems and Software Engineering*, 1(1):79–88, 2005. *cité page 72*

[SvGB05] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705–754, 2005. *cité page 14*

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997. *2 citations pages 5 et 20*

[TCPB07] Pablo Trinidad, Antonio Ruiz Cortés, Joaquín Peña, and David Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 51–56, 2007. *cité page 50*

[TCW+04] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White. A Multi-Agent Systems Approach to Autonomic Computing. In *Proceedings of the 3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004) – Volume 1*, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society. *2 citations pages 67 et 73*

[Tia03] Huaglory Tianfield. Multi-agent autonomic architecture and its application in e-medicine. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2003)*, pages 601–604. IEEE/WIC/ACM, 2003. *cité page 72*

[TLR+09] Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. WComp middleware for ubiquitous computing: Aspects and composite event-based Web services. *Annals of telecommunications*, 64(3-4):197–214, 2009. *cité page 33*

[Tou10] Lionel Touseau. *Politique de Liaison aux Services Intermittents dirigée par les Accords de Niveau de Service*. Thèse, Université Joseph-Fourier – Grenoble I, May 2010. *cité page 28*

[TR03] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+: Defining and Using Domain-specific Modeling Languages and Code Generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 92–93, New York, NY, USA, 2003. ACM. *cité page 51*

[UPn08] UPnP Device Architecture 1.0, 2008. *cité page 27*

[VMT11] Norha Machado Villegas, Hausi A. Müller, and Gabriel Tamura. On Designing Self-Adaptive Software Systems. *Sistemas y Telemática*, 9, num 18:29–51, 2011. *cité page 5*

[WB97] Mark Weiser and John Seely Brown. The Coming Age of Calm Technology. In *Beyond Calculation*, Lecture Notes in Computer Science, pages 75–85. Springer New York, 1997. *cité page 2*

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991. *2 citations pages 2 et 3*

[WJ95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10:115–152, June 1995. *cité page 67*

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999. *2 citations pages 37 et 44*

[WSG+13] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. On Patterns for Decentralized Control in Self-Adaptive Systems. In Rogério Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science*, pages 76–107. Springer Berlin Heidelberg, 2013. *cité page 73*

[WTKD04] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 70–77, 2004. *cité page 75*

[WWW+06] Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago Santiago. Quality of Service (QoS) Contract Specification, Establishment, and Monitoring for Service Level Management. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW 2006)*, pages 49–49, 2006. *cité page 29*

[Yu10] Jianqi Yu. *Ligne de produits dynamique pour les applications à services*. Thèse, Université Joseph-Fourier – Grenoble I, June 2010. *cité page 37*

[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. *cité page 93*

[ZFA14] Thierry Zylberberg, Nadia Frontigny, and Thepaut André. La révolution du Bien vieillir en France: Les technologies numériques au cœur de la transformation de l'action sociale. *Revue de l'Electricité et de l'Electronique*, (1):29–35, 2014. *cité page 177*