| 0 | 7 | P | A | U | U | 3 | 0 | 1 | 2 |

THESE

présentée à

L'UNIVERSITE DE PAU ET DES PAYS DE L'ADOUR

ECOLE DOCTORALE DES SCIENCES EXACTES ET DE LEURS APPLICATIONS

par

Fabien ROMEO

POUR OBTENIR LE GRADE DE

DOCTEUR

Spécialité : INFORMATIQUE

ADMINISTRATION DE COMPOSANTS LOGICIELS POUR SYSTEMES SANS FIL

Soutenue publiquement le :

14 novembre 2007

Après avis de :

Mme. ISSARNY Valérie	Directrice de Recherche, INRIA Rocquencourt (Rapporteur)
M. STEFANI Jean-Bernard	Directeur de Recherche, INRIA Rhône-Alpes (Rapporteur)

Devant la commission d'examen formée des rapporteurs et de :

Mme. GERVAIS Marie-Pierre	Professeur, Université Paris X Nanterre (Présidente)
M. PEZZE Mauro	Professeur, Université de Milano-Bicocca, Italie et Université de Lugano, Suisse
M. BLANC Xavier	Maître de Conférences, Université de Paris 6
M. BARBIER Franck	Professeur, Université de Pau et des Pays de l'Adour
M. BRUEL Jean-Michel	Maître de Conférences HDR, Université de Pau et des Pays de l'Adour

-2007-

Remerciements

La thèse est certes le fruit d'un travail personnel, mais elle est avant tout faite de rencontres et d'échanges. De nombreuses personnes ont contribué de près ou de loin, peut-être sans s'en rendre compte, à l'achèvement de cette thèse. Je tiens à les en remercier ici.

Tout d'abord, je remercie Valérie Issarny, Directrice de Recherche à l'INRIA Rocquencourt, et Jean-Bernard Stefani, Ingénieur Général des Télécommunications détaché comme Directeur de Recherche à l'INRIA Rhône-Alpes, de m'avoir fait l'honneur de rapporter ma thèse.

Je remercie ensuite Marie-Pierre Gervais, Professeur à l'Université Paris X Nanterre, et Xavier Blanc, Maître de Conférences à l'Université de Paris 6, d'avoir accepté de faire partie de mon jury de thèse.

In particular, I would like to thank Mauro Pezzè for his presence and his participation in my Ph.D committee and even more for giving me the chance to follow my research work with him at *l'Università degli Studi di Milano Bicocca*. Grazie Mille !

Un grand merci à Franck Barbier, mon directeur de thèse, pour avoir cru en moi dès la licence et pour tout ce qu'il m'a permis de faire par la suite. Au cours de mes années de doctorat, j'ai énormément appris à ses côtés et à tous les niveaux : technique, scientifique, pédagogique, voire même politique et stratégique.

Je remercie également Jean-Michel Bruel, qui a co-encadré mon travail de thèse, de m'avoir donné le goût de l'algorithmique et de l'informatique en général, lorsqu'il fut mon tout premier enseignant dans cette discipline. Merci de m'avoir « coaché » véritablement jusqu'au bout de cette thèse.

J'adresse toute ma gratitude à Nicolas pour avoir ouvert la voie (même si d'habitude je préfère faire la trace). Travailler avec lui a été un plaisir et je le remercie pour sa sympathie, sa gentillesse et pour l'ami qu'il est devenu. Je te souhaite tout ce qu'il y a de mieux à toi et ta petite famille, qu'elle ne cesse de s'agrandir !

Je tiens aussi à remercier tout le département informatique de l'UPPA. Notamment et en désordre alphabétique, Nabil et Annig pour leurs conseils éclairés, Laurent pour son assistance technique toujours efficace (mais pas que pour ça), Bruno pour son esprit « ouvert », ou plutôt « libre » c'est mieux, Régine et Yvette pour leur gentillesse, Sophie et Eric pour leur sympathie et leur bon esprit. Mes remerciements vont également à Alain pour sa simplicité, Amrane pour m'avoir permis de donner mes premiers cours magistraux, Congduc, Mauro, et Wilfrid pour les échanges toujours enrichissants que nous avons eus.

Je n'oublie évidemment pas de remercier mes collègues thésards Julien, Julien (et Jihene), Pierre (et Patricia), Cyril, Natacha et les non-thésards Laurent (encore) et Eric toujours de la partie, pour le basket, les hélicos, le poker, le vtt nocturne, le Garage, les soirées, les mentos et tout ce qu'on a pu faire ensemble, bref pour avoir été des vrais potes. Merci également aux anciens, Pierre, Mahmoud, Florent, Moussa, et Fatou pour à peu près les mêmes choses, mais avant.

Merci à tous ceux qui m'ont permis de m'aérer l'esprit sur les pentes enneigées et que je n'aurais pas encore cités, je veux parler de Jérémie, Xavier évidemment, Eric Andonoff, Charly et Marie, et toute la bande va{ski|roul|picol}er.

Un remerciement tout particulier à Kris (et Emeric) pour l'amitié que l'on a su conserver malgré la distance, et également pour m'avoir fait rencontrer Audrey. Merci.

Merci aussi à Niko pour toutes nos discussions autour d'un thé et pour toutes les destinations improbables où nous avons atterri, par fantaisie. Tu me surprendras toujours, mais ça, ça ne me surprend plus !

Un grand merci également à toute ma famille qui m'a toujours encouragé. Je ne remercierai jamais assez mes parents pour m'avoir toujours supporté dans mes choix, pour tout ce qu'ils ont fait et feront encore pour moi, pour leur amour tout simplement.

Enfin, mille mercis à Audrey, qui m'est apparue au cours de cette thèse pour illuminer ma vie. Je sais les efforts et les sacrifices que la thèse t'a également demandés et si j'ai tenu bon jusqu'au bout c'est bien grâce à ta patience, ton soutien et ton amour. Je t'aime.

À Audrey, et
À mes parents

Table des matières

Table des figures **ix**

Introduction **1**

Chapitre 1 Composants logiciels et systèmes sans fil **5**

1.1 Ingénierie des composants logiciels 6

1.1.1 Notion de composant logiciel 6

1.1.2 Modèles de composants 11

1.1.3 Infrastructure d'accueil de composants 19

1.2 Ingénierie des composants logiciels dans les systèmes sans fil 22

1.2.1 Caractéristiques des systèmes sans fil 22

1.2.2 Modèles de composants spécifiques au sans fil 27

1.3 Du besoin d'administration pour les composants sans fil 30

Chapitre 2 Administration d'applications logicielles **33**

2.1 Concepts et principes 34

2.1.1 Définition générale de l'administration 34

2.1.2 Le modèle de gestion OSI 35

2.1.3 Activités de l'administration 37

2.2 Approches classiques 40

2.2.1 Approches centralisées 41

2.2.2 Approches hiérarchiques 42

2.2.3 Approches par politique 43

2.2.4 Approches autogérées 44
2.3 Administration des composants logiciels 46
2.3.1 Approches basées sur l'infrastructure 46
2.3.2 Approches basées sur l'architecture 54
2.3.3 Approches basées sur le comportement 58
2.4 Solutions d'administration des composants logiciels sans fil 62
2.4.1 OMA Device Management 62
2.4.2 JSR 232 & JSR 233 66
2.4.3 Dynamically Programmable and Reconfigurable Software 68
2.5 Synthèse 71

Chapitre 3 Conception de composants logiciels en vue de leur administration 73

3.1 Présentation générale de l'approche 74
3.2 Modélisation de composants UML 2 pour les systèmes sans fil 75
3.2.1 Introduction à UML 2 77
3.2.2 Structure : modèle de composants UML 2 79
3.2.3 Comportement : machines à états UML 2 82
3.3 PauWare.Velcro : exécutabilité et persistance à l'exécution de machines à états UML 2 87
3.3.1 MDA et Executable UML 87
3.3.2 Principes de PauWare.Velcro 88
3.3.3 Principes du modèle de composants PauWare 95
3.4 Synthèse 100

Chapitre 4 Une architecture d'administration de composants logiciels sans fil 101

4.1 Architecture d'administration 102
4.1.1 Les architectures possibles 102
4.1.2 L'architecture retenue 105
4.2 Administration interne 106
4.2.1 Gestionnaire interne et composant métier 107

4.2.2 Capteurs et effecteurs internes 108
4.3 Administration externe 111
4.3.1 Réplication des machines à états 111
4.3.2 Capteurs et effecteurs externes 114
4.3.3 Contrôle à distance 118
4.4 Exemple d'une politique pour la gestion de compositions 119
4.4.1 Composition comportementale avec les *Statecharts* et la relation Tout-Partie 119
4.4.2 Une politique d'administration pour assurer la composition rigoureuse du comportement 121
4.5 Synthèse 124

Chapitre 5 WMX : administration Java de composants logiciels sans fil 125

5.1 L'implémentation de la réflexion dans PauWare.Velcro : un pré-requis 126
5.1.1 Utilisation de la réflexion Java dans PauWare 126
5.1.2 Implémentation d'un mécanisme remplaçant la réflexion pour Java ME . 127
5.1.3 Génération de code 131
5.2 Mise en œuvre de l'architecture avec des technologies Java 131
5.2.1 Implémentation des composants WMX 132
5.2.2 Ports de communication sans fil et système d'administration à distance . 134
5.3 Visualisation du comportement 135
5.3.1 La console JMX 135
5.3.2 L'outil PauWareView 137
5.4 Etude de performance 137
5.4.1 Elaboration du banc d'essai et mesures 139
5.4.2 Analyse des résultats 140

Conclusion et perspectives 143

Bibliographie 147

Table des figures

1.1 Vue externe d'un composant de forme canonique 7

1.2 Exemple de composition horizontale 8

1.3 Exemple de composition verticale 8

1.4 Vue externe d'un composant de forme canonique prenant en compte la configuration 9

1.5 Architecture détaillée de l'assemblage d'un composant `Weather Center` avec un composant `Wind Sensor` 17

1.6 Exemple d'un composant Fractal 18

1.7 Cycle de vie d'un composant 21

1.8 Hétérogénéité des appareils sans fil 23

1.9 Java et ses différents domaines d'application 25

1.10 Correspondance entre les configurations J2ME et J2SE 25

2.1 Architecture fondamentale de l'administration 35

2.2 Le protocole SNMP 37

2.3 Cycle de vie d'une ressource administrée 39

2.4 Le modèle gestionnaire - agent 40

2.5 Approche d'administration centralisée 41

2.6 Approche d'administration hiérarchique 42

2.7 Approche d'administration par politique 43

2.8 Approche d'administration autogérée 45

2.9 Architecture de JMX 48

2.10 Unités d'administration des serveurs Java EE (JSR 77) 51

2.11 Interface `StateManageable` du JSR 77 52

2.12 Le modèle des états administrables du JSR 77 53

2.13 Architecture classique de l'administration des EJB 2.x avec JMX 53
2.14 Conception réflexive de DYVA 55
2.15 Architecture de DYVA 56
2.16 Modélisation des ressources dans SAFRAN 57
2.17 Architecture générale des approches basées sur le comportement 59
2.18 Architecture d'OMA DM 63
2.19 Exemple de *DM tree* 64
2.20 *OMA DM Firmware Update MO* 65
2.21 Exemple de commande *OMA DM Replace* 65
2.22 Cycle de vie des composants avec livraison dans SCOMO 66
2.23 *Application Manager* dans le JSR 232 67
2.24 Micro Building Blocks 69
2.25 Action interprétée dans les composants DPRS 69
2.26 Externalisation dans les composants DPRS 70

3.1 Exemple de comportement utilisant des états abstraits de la variable `i` 75
3.2 De la modélisation à l'administration : conception d'un composant logiciel administrable 76
3.3 Exemple des quatre niveaux de modélisation d'UML 80
3.4 Métamodèle des composants UML 81
3.5 Forme canonique d'une transition dans un diagramme d'états-transitions 83
3.6 Exemple de machine à états simple 84
3.7 Exemple de machine à états avec avec états hiérarchiques 84
3.8 Exemple de machine à états avec états orthogonaux 85
3.9 Forme canonique d'une transition générant un signal dans un *Statechart* 86
3.10 La machine à états PauWare 90
3.11 Machine à états du comportement d'une lampe `Light` 91
3.12 Trace d'exécution avec traitement de l'événement `turnOff` avant l'événement `turnOn` au temps t6 (l'événement `turnOff` est pris en compte) 94
3.13 Trace d'exécution avec traitement de l'événement `turnOff` après l'événement `turnOn` au temps t6 (l'événement `turnOff` n'a pas d'effet) 94
3.14 Un composant canonique PauWare intègre une machine à états pour réaliser son comportement 95

3.15 Implémentation du composant PauWare 96
3.16 Comportement du composant `PauWare Component` 97
3.17 Relations et comportements des composants `Controller` et `Gate` 99
3.18 Comportement d'un composant composite dans PauWare 99
3.19 Un composant PauWare administrable 100

4.1 Un programme distribué vu comme un programme réactif 103
4.2 Architecture avec logique du comportement distante 103
4.3 Architecture avec logique du comportement embarquée 105
4.4 L'architecture retenue 106
4.5 Structure interne d'un composant WMX 107
4.6 Architecture détaillé du composant `WMX Component` 109
4.7 Comportement du composant `WMX Component` 110
4.8 Les trois étapes de la synchronisation par les événements 112
4.9 Problème de l'évaluation des gardes avec la synchronisation par les événements . 113
4.10 Les trois étapes de la synchronisation par les transitions 115
4.11 Architecture détaillée du port d'administration qui relie les gestionnaires interne et externe d'un composant WMX 116
4.12 Echange de messages entre un composant et sa réplique lors de l'activité d'observation 117
4.13 Echange de messages entre un composant et sa réplique lors de l'activité de contrôle 118
4.14 Comportement du composant `Light` 120
4.15 Comportement du composant `TrafficLight` 121
4.16 Comportement du `CompositeManager` 122
4.17 Comportement du `PartManager` 123

5.1 Architecture de PauWare pour Java ME et Java SE 130
5.2 Interface `Statechart_monitor_listener` 132
5.3 Implémentation du gestionnaire du composant `Light` 135
5.4 Administration d'un composant WMX via la console JMX 136
5.5 Administration via une console JMX et surveillance via le PauWareView 138

Introduction

Contexte de la thèse

La génération Internet est là ![1] L'utilisation d'applications logicielles s'est démocratisée et n'intervient plus seulement dans le cadre d'une activité professionnelle, mais elle s'intègre également dans la vie de tous les jours, au point d'être perçue comme banale pour certains, car tellement indispensable. Ce phénomène s'est encore amplifié avec l'essor des communications sans fil et la miniaturisation des appareils. En effet, les systèmes sans fil, tels que les PDAs, les téléphones sans fil, ou encore les ordinateurs de bord dans les automobiles, permettent aujourd'hui d'avoir accès en permanence à un grand nombre d'applications : messagerie électronique, traitement de texte, système de navigation GPS, application photo, vidéo, etc. La demande est forte mais le secteur demeure concurrentiel. L'industrie logicielle doit ainsi faire face à une exigence de plus en plus grande en termes de réduction des coûts et des délais de production du logiciel.

A cet égard, le paradigme des composants logiciels est une approche de développement d'applications particulièrement intéressante. Dans le but de maximiser la réutilisation du logiciel, ce paradigme considère une application comme un assemblage de composants logiciels pré-fabriqués et non comme une entité monolithique à créer *in extenso*. Cela permet d'améliorer notamment la réactivité aux besoins des utilisateurs en offrant des possibilités d'adaptation des applications par l'ajout ou le remplacement de composants ciblés.

Cette pratique, qui a été popularisée par le concept de *plugin*, est particulièrement présente dans les systèmes sans fil pour répondre aux attentes des utilisateurs *on demand*. En effet, il n'est pas rare aujourd'hui que les utilisateurs de systèmes sans fil téléchargent des composants pour ajouter ou remplacer telle ou telle fonctionnalité à leur téléphone portable, à leur PDA, ou à d'autres dispositifs.

Dans ce contexte, les ajouts et les remplacements successifs peuvent nuire à la prédictibilité du comportement de l'assemblage constituant l'application. En effet, lors du développement, l'impossibilité de déterminer avec précision les composants qui formeront l'application, augmente les possibilités de dysfonctionnement à l'exécution. Dans le cas des systèmes sans fil, l'hétérogénéité des environnements de déploiement augmente encore les sources d'erreurs. Ce problème

[1] « *The Net Generation has arrived !* » [155]

fait apparaître le besoin d'administration des composants logiciels à l'exécution afin de détecter les comportements anormaux et les corriger le cas échéant.

Objectif de la thèse

Dans cette thèse, nous nous sommes intéressés à l'administration du comportement des composants logiciels et son application aux environnements sans fil. Pour mener notre recherche, nous nous sommes fixés les objectifs suivants :

- capitaliser le travail de modélisation effectué lors du développement des composants pour les besoins de l'administration. Les modèles du comportement des composants doivent pouvoir être utilisés directement lors de leur administration,
- élever le niveau d'abstraction de l'information d'administration afin de définir des politiques d'administration basées sur les états de fonctionnement des composants,
- concevoir une approche réellement applicable pour l'administration des composants dans les systèmes sans fil.

Afin de remplir ces objectifs, l'approche que nous avons retenue s'appuie sur PauWare, une technologie issue de recherches précédentes sur l'ingénierie dirigée par les modèles. PauWare permet l'exécution de modèles UML 2 et en particulier des machines à états. Nous avons déjà pu expérimenter avec succès les fonctionnalités de PauWare dans le cadre du test intégré de composants logiciels [22]. Nous proposons de partir de ces travaux pour définir une approche d'administration qui permette de concevoir des composants logiciels administrables au niveau du système sans fil et les gestionnaires *ad hoc* au niveau du système d'administration.

En termes de réalisations, notre contribution comprend les technologies et les outils logiciels suivants :

- PauWare.Velcro qui rend possible l'utilisation des machines à états exécutables PauWare dans les systèmes sans fil,
- WMX (*Wireless Management eXtensions*) qui met en œuvre l'architecture d'administration que nous avons conçue,
- un outillage complet constitué de consoles d'administration distantes, d'un outil de visualisation graphique, de différents adaptateurs de communication sans fil.

Organisation du document

Le contenu de ce document est organisé en cinq chapitres. Dans le premier chapitre, nous présentons l'ingénierie logicielle basée composants et son application au développement d'applications déployées dans des systèmes sans fil. Le deuxième chapitre dresse l'état de l'art que nous avons réalisé sur les différentes approches d'administration et en particulier sur les solutions existantes pour les systèmes basés composants, si possible en environnement sans fil. Le troisième chapitre expose notre approche adaptée aux systèmes sans fil pour la conception de

composants logiciels administrables. Dans le quatrième chapitre, nous présentons ensuite l'architecture d'administration que nous avons conçue et qui tire profit de notre approche de conception des composants. Enfin, le cinquième chapitre décrit WMX (*Wireless Management eXtensions*), l'implémentation réalisant notre architecture d'administration, et présente une étude évaluant ses performances.

Chapitre 1

Composants logiciels et systèmes sans fil

Sommaire

1.1 Ingénierie des composants logiciels 6

1.1.1 Notion de composant logiciel 6

1.1.2 Modèles de composants 11

1.1.3 Infrastructure d'accueil de composants 19

1.2 Ingénierie des composants logiciels dans les systèmes sans fil 22

1.2.1 Caractéristiques des systèmes sans fil 22

1.2.2 Modèles de composants spécifiques au sans fil 27

1.3 Du besoin d'administration pour les composants sans fil 30

L'ingénierie des composants logiciels, ou en anglais *component-based software engineering* (CBSE), est un paradigme de développement qui vise à accélérer le développement du logiciel et à en réduire les coûts de production en le construisant par un assemblage de composants préfabriqués [99]. La conception, le développement et la maintenance des composants logiciels et des applications à base de composants sont certes des processus complexes mais qui apportent des bénéfices significatifs dans différents domaines d'applications. C'est en particulier le cas dans les systèmes sans fil dont la partie logicielle prend de plus en plus d'importance.

Suite à la rapide évolution du matériel en termes de communication et de performance, les systèmes sans fil se sont complexifiés et de nombreuses approches tirent aujourd'hui profit du paradigme des composants logiciels pour développer des applications embarquées dans ces systèmes. Les composants logiciels doivent ainsi faire face à de nouveaux défis spécifiques aux systèmes sans fil.

Ce chapitre a pour but d'introduire les principaux concepts de l'ingénierie des composants logiciels ainsi que les avancées en recherche sur les composants logiciels dans les applications sans fil. Nous préciserons également le champ des applications sans fil que nous considérons dans nos travaux.

1.1 Ingénierie des composants logiciels

Cette section présente les principes du CBSE à travers la notion de composant logiciel et présente ensuite les concepts techniques propres aux technologies logicielles à base de composants.

1.1.1 Notion de composant logiciel

Si l'idée de composant logiciel est l'essence même du CBSE, sa définition est sujette à de nombreuses propositions différentes. Citons par exemple [7, 8, 39, 71, 103, 154]. Les nuances entre les diverses définitions sont toutefois assez subtiles mais celle de Szyperski [154], la plus reprise dans la littérature, semble faire consensus.

Composant logiciel

> *« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties. »* [154]

Cette définition a été établie à partir des caractéristiques communes des définitions de l'époque lors du *Workshop on Component-Oriented Programming* à ECOOP'96. Nous nous appuyons sur cette définition pour aborder les différentes notions que l'on retrouve dans l'ingénierie des composants logiciels.

Encapsulation

Un composant est défini comme une unité de composition. Il représente une partie d'un système logiciel qui encapsule des fonctionnalités spécifiques. Selon le principe de l'*information hiding* [120], l'encapsulation permet de cacher aux utilisateurs extérieurs, *i.e.* d'autres composants, les détails d'implémentation des fonctionnalités du composant. Toute communication avec l'extérieur est alors réalisée à travers des interfaces bien définies. Ainsi le composant est vu par ses utilisateurs comme une boîte noire et seuls sont accessibles les services qu'il expose dans ses interfaces de services fournis.

Interfaces de services fournis

Une interface de services fournis est le seul moyen par lequel on peut manipuler un composant. Elle caractérise complètement le composant vis-à-vis de son client en spécifiant l'ensemble des services qu'il met à sa disposition. Cette interface permet d'établir un faible couplage [165, p. 84–104] entre le composant et son client en empêchant ce dernier d'accéder directement à l'implémentation du composant. On obtient ainsi une composition propre du composant favorisant sa substituabilité. Un composant disposant de la même interface pourra donc remplacer le premier composant. Cela est généralement le cas des différentes versions d'un composant.

Interfaces de services requis

Duale à la notion d'interface de services fournis, une interface de services requis permet à un composant de déclarer les services externes dont il a besoin pour fonctionner. Afin d'éviter un fort couplage et de ne pas nuire à sa composabilité, c'est seulement par cette interface que le composant peut communiquer avec son environnement extérieur. Les dépendances fonctionnelles du composant sont ainsi clairement explicitées et regroupées, ce qui permet de contenir la cohésion du composant [165, p. 105–142] pour en favoriser la réutilisation.

Composant canonique

Les notions que nous avons vues jusqu'ici nous permettent de déterminer d'une manière générale la perception que possède l'environnement extérieur sur les composants : un composant est une boîte noire qui expose ses fonctionnalités via une interface de services fournis et exprime ses dépendances via une interface de services requis. Le diagramme de composant UML 2 (*Unified Modeling Language*) de la figure 1.1 représente la vue externe d'un composant de forme canonique.

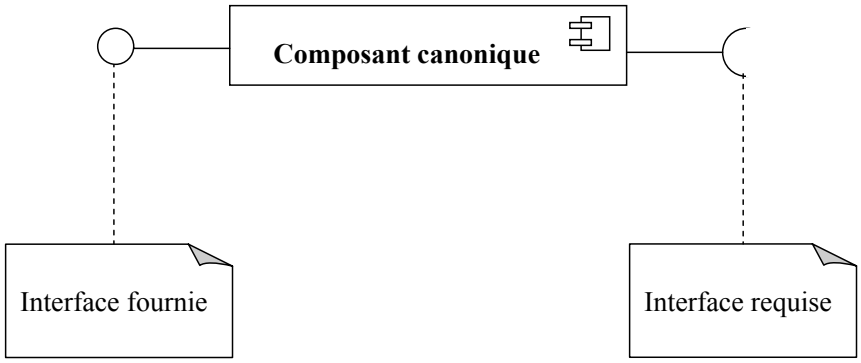


Fig. 1.1 – Vue externe d'un composant de forme canonique

Composition

Par définition, un composant est fait pour être composé. On obtient un assemblage ou une composition de composants en interconnectant leurs interfaces de sorte que les composants demandant des services, soient reliés aux composants offrant ces dits services. Cette composition peut être réalisée par un tiers, c'est-à-dire une personne qui n'a pas réalisé le composant. La capacité d'autodescription du composant est alors critique : il doit être délivré sous une forme qui permette d'appréhender correctement ses fonctionnalités.

Le résultat d'un assemblage peut constituer un nouveau composant de granularité supérieure : on parle alors de composant composite. Les composants inclus dans un tel composant

sont nommés sous-composants ou plus simplement composés. Cette composition particulière est principalement connue sous le nom de composition hiérarchique. Elle est également appelée verticale [16] par opposition à la composition horizontale dans laquelle les composants assemblés sont considérés au même niveau de granularité (cf. figures 1.2 et 1.3).

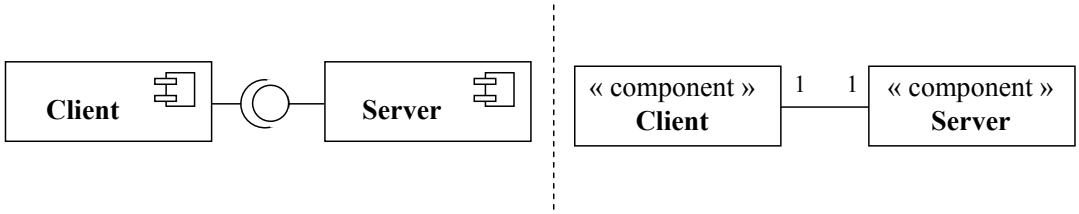


FIG. 1.2 – Exemple de composition horizontale

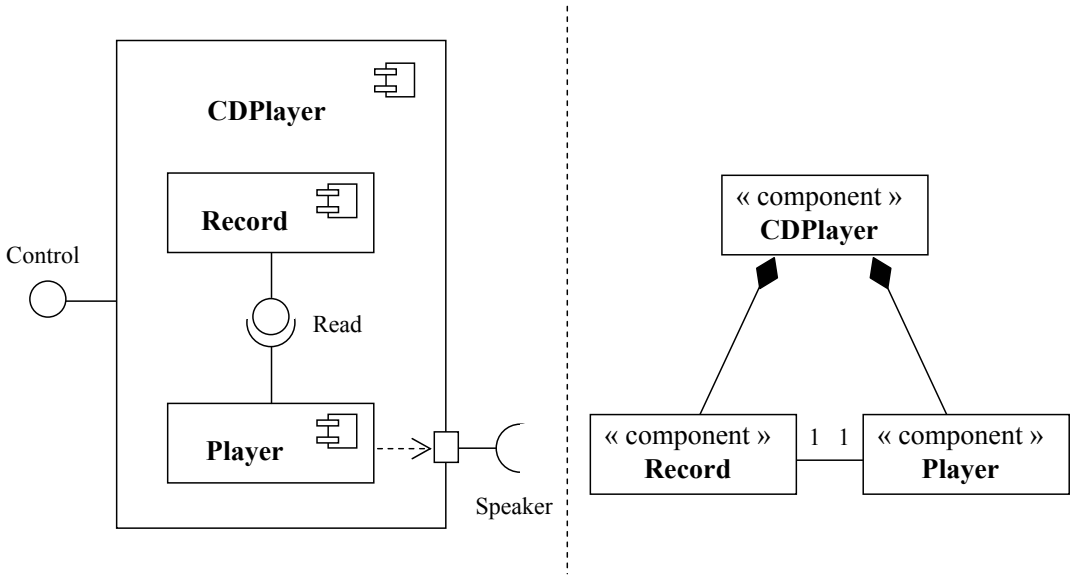


FIG. 1.3 – Exemple de composition verticale (inspiré de [17])

Déploiement

Un composant est une unité de composition mais c'est également une unité de déploiement. La phase de déploiement [61] consiste à mettre en place le composant dans un environnement cible et à vérifier son aptitude à fonctionner correctement avec cet environnement. Une application à base de composants ne peut effectivement démarrer que lorsque tous les composants qui la constituent sont déployés [63]. Du fait des nombreuses dépendances et interdépendances qu'il peut exister au sein d'un système à composants, la phase de déploiement peut mener à des vérifications poussées.

Le principal aspect du déploiement est le passage du composant de sa forme inerte et descriptive à sa forme exécutable dans un environnement cible. Or, la réutilisation amène à déployer les composants dans des environnements hétérogènes. Afin de pouvoir adapter les composants à cette hétérogénéité, ils sont généralement dotés d'une interface spécifiquement dédiée à leur configuration. Nous pouvons ainsi affiner notre représentation d'un composant de forme canonique (cf. figure 1.1) avec la figure 1.4.

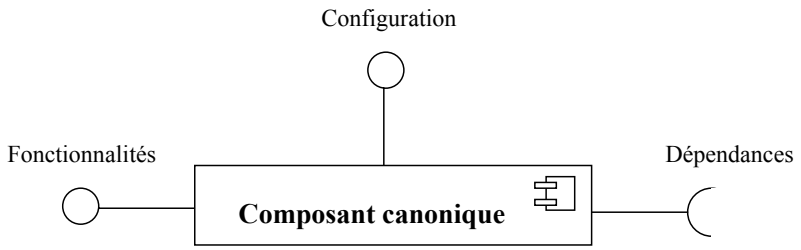


Fig. 1.4 – Vue externe d'un composant de forme canonique prenant en compte la configuration

Composants et objets

Les liens qui existent entre les notions de composant et d'objet ont souvent été débattus et différents points de vue s'affrontent sur le sujet [64]. Selon Szyperski [154], un composant correspond à un élément statique qui capture une partie de l'invariant du système alors qu'un objet est un élément changeant qui correspond plus à la dynamique du système. Il voit ainsi un composant comme une unité de déploiement et un objet comme une unité d'instanciation. Un composant peut être décrit par une collection de classes fortement interdépendantes dont l'accès depuis l'extérieur du composant est strictement spécifié par les interfaces du composant. Déployé sous forme binaire, le composant prend alors vie par l'instanciation de ses objets. Cependant ce n'est qu'une possibilité, les composants pouvant tout aussi bien être réalisés par une approche procédurale, une approche fonctionnelle, ou toute autre approche.

Au contraire, Meyer voit les composants comme une évolution naturelle de l'approche orientée objet [103, 104]. Pour lui, composants et objets partagent les mêmes principes fondamentaux dans le but de permettre la réutilisation dans le développement du logiciel. Par exemple, le principe d'encapsulation que l'on retrouve dans les composants est une caractéristique majeure des classes d'objets. On peut alors se demander si une classe est véritablement un composant ou non.

Pour D'Souza et Wills [46], une classe peut constituer un composant si elle est délivrée avec les interfaces qu'elle requiert et qu'elle implémente. Ils font également les distinctions suivantes entre objets et composants :

- les composants utilisent souvent des mécanismes de stockage persistant tels qu'une base de données ou un système de fichiers alors que les objets ont un état local,

- les composants peuvent avoir différents moyens de communications alors que les objets qui sont généralement limités à l'envoi de messages,
- les composants forment souvent des unités de granularité plus importante que les objets,
- les interfaces des composants ne se limitent pas à une liste de méthodes mais peuvent fournir des informations supplémentaires via les propriétés ou les événements par exemples.

Malgré les différences qu'il existe entre composants et objets et le fait qu'en théorie les composants peuvent être construits avec n'importe quelle approche, les modèles de composants qui connaissent le plus de succès aujourd'hui sont tous construits à partir de technologies orientées objet. Dans la pratique donc, les composants et les objets coexistent tout en restant des concepts séparés.

Composants et services

L'importance croissante des services Web, et plus généralement de l'architecture orientée service (AOS)[2], tend à instaurer la notion de service comme la nouvelle brique de base de construction d'application. Une AOS est une infrastructure qui rend disponible des services métier indépendants qui peuvent être utilisés et composés dynamiquement dans des applications clientes ou d'autres services de plus haut niveau [66]. Le besoin de faire collaborer les systèmes d'information de différentes organisations à travers les réseaux, et en particulier Internet, a créé un engouement autour de ce concept qui a engendré une vague de nouveaux standards tels que XML Schema, SOAP (*Simple Object Access Protocol*), UDDI (*Universal Description, Discovery and Integration*), WSDL (*Web Services Definition Language*), ou WSFL (*Web Services Flow Language*) pour permettre l'interopérabilité des systèmes.

Pour Szyperski, les services ne sont pas le logiciel lui-même mais sont délivrés à travers le logiciel par leur fournisseur sur une infrastructure :

> *« Services are different from components in that they require a service provider. A service is an instance-level concept — where such instances can be component instances. These instances are "live" and thus require grounding in concrete hardware, software, and organizational infrastructure. »* [154]

Il est donc nécessaire de trouver les artefacts logiciels permettant d'incarner les services. Les propriétés des composants logiciels en termes de déploiement, de composition et de communication entre autres, en font des candidats de choix. Ainsi des technologies à composants telles que Java EE ou .NET permettent de publier leurs composants en tant que services Web. L'initiative récente du Service Component Architecture (SCA) [15] propose maintenant un modèle d'architecture à composant pour la conception d'application orientée service.

[2]De l'anglais *Service-Oriented Architecture* (SOA).

Composants et ADL

Les langages de description d'architecture ou ADL (*Architecture Description Languages*) utilisent également la notion de composant. Ces langages permettent de décrire l'architecture d'un système logiciel dont Nierstrasz et Meijler donnent la définition suivante :

> *« A (software) architecture is a description of the way in which a specific system is composed from its components. »* [109]

Une architecture doit donc représenter l'ensemble des composants d'un logiciel et leurs interconnections. Pour cela, les ADL distinguent les concepts de *composant*, *connecteur* et *configuration* [57, 100]. Les composants sont des unités d'abstraction qui possèdent des interfaces. Ces interfaces, souvent appelées ports, définissent des points d'interaction entre le composant et le monde extérieur. Les connecteurs permettent de lier les composants entre eux par leurs interfaces en définissant leurs interactions. Une configuration représente alors un ensemble de composants reliés par des connecteurs de façon à décrire le système à un certain niveau d'abstraction. Les ADL prennent souvent en compte la composition hiérarchique, ce qui permet à un composant d'être lui-même décrit par une configuration entière.

Les concepts et principes de haut niveau des ADL sont particulièrement intéressants dans les premières phases du cycle de vie du logiciel. Le fait de modéliser l'assemblage de la future application permet de raisonner sur les propriétés du système afin d'en assurer un développement rigoureux. Ils sont en fait complémentaires des modèles de composants qui, au contraire, se focalisent plus sur les phases d'implémentation, de déploiement et d'exécution.

1.1.2 Modèles de composants

Selon les principes du CBSE, il est possible de développer une application en assemblant des composants préfabriqués provenant de fournisseurs différents. Pour mener à bien un tel développement, il est nécessaire que le concepteur d'application à base de composants puisse effectivement interconnecter les différents composants sélectionnés pour composer l'application. Il est donc crucial que chaque parti (concepteurs et fournisseurs) s'accorde sur un modèle de composants, c'est-à-dire un standard commun permettant la manipulation uniforme des composants et leur interopérabilité.

> *« A component model specifies the standards and conventions imposed on developers of components. »* [8]

Un modèle de composants établit donc une norme définissant un ensemble de règles et de contraintes techniques imposées sur la construction et l'utilisation des composants. Pour réaliser son application, le concepteur peut alors s'appuyer sur le modèle de composants utilisé par son application afin d'établir des hypothèses sur les composants qu'il réutilise.

Le choix d'une technologie à composants est fortement dépendant de l'adéquation entre le modèle de composants sous-jacent et l'application ciblée. Ainsi, la construction d'interfaces

homme-machine sera simplifiée par l'utilisation de JavaBeans qui appliquent les principes du modèle MVC (*Model-View-Controller*) ; les applications 3-tiers tireront profit de l'architecture définie par Java EE (*Enterprise Edition*) ou par CCM (*Corba Component Model*) ; le développement d'applications sur des appareils sans fil sera facilité par l'adoption de Java ME (*Micro Edition*).

Afin d'avoir un aperçu concret de ce qu'est un composant, nous allons présenter les modèles de composants JavaBeans, EJBs, Spring et Fractal.

JavaBeans, un modèle simple

Sun Microsystems a mis au point le modèle de composants JavaBeans en particulier pour la construction d'interfaces graphiques utilisateur (GUI). Sa simplicité et son efficacité en font cependant un modèle de choix pour de nombreuses autres applications. La spécification des JavaBeans nous donne la définition suivante :

> *« A Java Bean is a reusable software component that can be manipulated visually in a builder tool. »* [144]

Cette définition met en avant le caractère réutilisable de ces composants, ce qui est clairement dans l'esprit de la définition de Szyperski. Un composant JavaBean n'est donc pas conçu pour fonctionner seulement dans une application en particulier. Il peut être composé par des tiers et doit même pouvoir l'être visuellement. Sun démontre les possibilités de son modèle de composants avec le logiciel Bean Builder qui se distingue notamment par :

- des mécanismes d'introspection qui décrivent les interfaces et les propriétés des composants,
- la possibilité de configurer manuellement les propriétés des composants,
- une communication événementielle permettant l'assemblage des composants grâce à un mécanisme de publication/abonnement de notifications,
- des mécanismes de persistance afin de conserver la configuration des composants entre de multiples exécutions.

Les JavaBeans sont donc explicitement conçus pour interagir dans deux contextes différents, au moment de la composition/configuration dans l'outil de construction et à l'exploitation réelle dans l'environnement d'exécution. Cela est principalement mis en œuvre grâce aux mécanismes de réflexion de la plate-forme Java (paquetage `java.lang.reflect.*`) qui permettent à l'exécution de retrouver la classe d'un objet, ses méthodes, ses attributs et de les manipuler.

Ainsi les JavaBeans sont de simples objets Java sur lesquels le modèle de composant impose un certain nombre de conventions de nommage pour spécifier leurs propriétés, leurs événements, ou encore leurs méthodes. Par exemple, une simple propriété est définie sur un composant par une méthode portant le nom de la propriété, préfixé par les termes `get` ou `set` selon que la propriété doit être accessible en lecture ou en écriture, conformément au *Design Pattern* [54] décrit par le listing 1.1.

```
// propriété simple en lecture seule
    public String getName();

// propriété simple en écriture seule
    public void setSize(int size);

// propriété simple en lecture/écriture
    public Time getTime();
    public void setTime(Time time);

// propriété booléenne
    public boolean isVisible();
    public void setVisible(boolean visibility);

// propriété indexée
    public Socket[] getConnections();
    public void setConnections(Socket connections[]);
    public Socket getConnections(int index);
    public void setConnections(int index, Socket connection);
```

LIST. 1.1 – *Design Pattern* des propriétés JavaBeans

Du point de vue du déploiement, un JavaBean se présente sous la forme d'un paquetage regroupant dans une archive JAR les classes d'implémentation du composant. Optionnellement, ce paquetage peut également comporter une classe dérivée de `java.beans.BeanInfo` afin de décrire le composant sans utiliser par la suite la réflexion et pour fournir des informations supplémentaires qui n'auraient pu être découvertes par réflexion. Le composant est alors décrit de façon programmatique.

Les JavaBeans constituent un modèle simple tout à fait approprié pour le développement d'applications clientes mais leur utilité trouve ses limites sur des projets de plus grande envergure induisant des architectures complexes et notamment distribuées.

Enterprise JavaBeans, un modèle d'entreprise

Les Enterprise JavaBeans (EJB) sont des composants conçus spécifiquement pour la construction de la partie serveur des applications d'entreprise impliquant des architectures 3-tiers (client / serveur / base de données). Depuis sa parution en 1998, le modèle des EJB a évolué à travers les différentes spécifications éditées par Sun et constitue aujourd'hui un standard dans l'industrie logicielle. En 2006, de nouvelles améliorations ont encore été apportées au modèle avec la version 3.0 [150]. Afin d'apprécier au mieux ces récentes évolutions, nous nous appuierons sur la spécification 2.1 [147] pour présenter le modèle puis nous évoquerons les principaux apports de la spécification 3.0.

Du point de vue de sa structure, un composant EJB possède deux points d'accès implémentés par des interfaces Java :

- son interface *Home* fournit un ensemble de services permettant à l'infrastructure, dans laquelle le composant est déployé, de gérer une partie de son cycle de vie (création, destruction, et également recherche dans le cas d'EJB persistants). Un client d'un composant EJB doit donc d'abord s'adresser à son interface *Home* pour déclencher la création ou la localisation du composant. La déclaration d'une telle interface doit suivre certaines règles spécifiées par le modèle des EJB, comme par exemple dériver de l'interface `javax.ejb.EJBHome` fournie par l'API de Java EE. Le listing 1.2 montre un exemple de déclaration d'une interface *Home* pour un composant `BankingSystem`.
- son interface *Remote* représente l'interface fonctionnelle par laquelle le composant déclare ses services d'application. Également appelés services métier, ces services sont ceux pour lesquels un client se connecte au composant. C'est donc par cette interface qu'un client utilise réellement le composant. Parmi d'autres conventions qu'elle doit suivre, cette interface doit notamment dériver de l'interface `javax.ejb.EJBObject`. Le listing 1.3 montre un exemple de déclaration d'une interface *Remote* pour un composant `BankingSystem`.

```
// Cycle de vie du composant BankingSystem
public interface BankingSystemHome extends javax.ejb.EJBHome {
    BankingSystem create() throws java.rmi.RemoteException,
                                  javax.ejbCreateException;
}
```

List. 1.2 – Interface *Home* du composant `BankingSystem`

```
// Méthodes métier du composant BankingSystem
public interface BankingSystem extends javax.ejb.EJBObject {
    void deposit(BigDecimal amount, Account account) throws
       java.rmi.RemoteException;
    void withdraw(BigDecimal amount, Account account) throws
       java.rmi.RemoteException;
    void transfer(BigDecimal amount, Account from, Account to)
        throws java.rmi.RemoteException;
}
```

List. 1.3 – Interface *Remote* du composant `BankingSystem`

Le modèle des EJB considère trois grandes catégories de composants permettant de traiter différents aspects serveur d'une application d'entreprise :

- les *Session Beans* sont des composants qui communiquent en mode synchrone avec leurs clients. Ces composants implémentent la classe `javax.ejb.SessionBean` et peuvent être de deux types, *Stateless* ou *Statefull* :
  - les *Stateless Session Beans* ne maintiennent pas d'état entre plusieurs appels successifs, ce qui leur permet de servir plusieurs clients en même temps.

  - les *Stateful Session Beans*, au contraire, peuvent conserver des informations spécifiques à un client entre deux appels successifs au cours d'une même session. Dans ce cas, les sessions sont traitées séquentiellement, autrement dit ces composants ne peuvent servir qu'un seul client à la fois.
- les *Entity Beans* sont des composants qui possèdent un état persistant. Ils modélisent des données métier qui peuvent être utilisées simultanément par plusieurs clients. Typiquement, un *Entity Bean* correspond à une table dans une base de données relationnelle et ses instances, qui possèdent un identificateur unique, *i.e.* une clé primaire, constituent les lignes de la table.
- les *Message Driven Beans* sont des composants qui communiquent en mode asynchrone. Ainsi, ils ne possèdent pas d'interfaces *Home* et *Remote* mais s'exécutent à la réception d'un message dans une file d'attente JMS (*Java Message Service*).

Le développement d'une application à base d'EJB n'est pas envisageable sans l'utilisation d'outils appropriés pour gérer au mieux la configuration des composants et automatiser les tâches d'implémentation les plus répétitives. La récente spécification EJB 3.0 tend à pallier cette lourdeur par l'utilisation d'annotations qui réduisent le caractère verbeux de l'implémentation des composants et offrent une meilleure séparation des préoccupations. Introduites dans la version 1.5 de Java, les annotations permettent d'ajouter des métadonnées au code source. Ces informations sont ensuite rendues accessibles à l'exécution par réflexion. Grâce à ce moyen, il n'est par exemple plus nécessaire de construire explicitement les interfaces *Home* et *Remote* d'un EJB. Il suffit maintenant d'annoter une simple interface Java avec le marqueur `@Remote` comme le montre le listing 1.4. De cette façon, l'interface fonctionnelle du composant est couplée plus faiblement à la structure même d'un composant EJB et peut être plus facilement réutilisée par ailleurs.

```
1 // Méthodes métier du composant BankingSystem
2 @Remote
3 public interface BankingSystem {
4     void deposit(BigDecimal amount, Account account);
5     void withdraw(BigDecimal amount, Account account);
6     void transfer(BigDecimal amount, Account from, Account to)
          ;
7 }
```

List. 1.4 – Interface métier du composant `BankingSystem`

D'une manière générale, les EJB s'orientent vers un modèle plus efficace et plus simple à utiliser. Ainsi, la nouvelle spécification propose dorénavant une alternative aux *Entity Beans*[3] sous la forme d'une API pour la persistance [151] reposant sur le concept d'*Entities*, c'est-à-dire des composants légers permettant de gérer des données persistantes mais n'utilisant pas de

[3]Les *Entity Beans* sont néanmoins conservés dans la même forme que dans la précédente spécification.

conteneur EJB comme les *Entity Beans*. L'évolution des EJB s'inspirent clairement des modèles légers comme Spring que nous étudions dans la section suivante.

Spring, un modèle léger

Démarré en 2003, Spring est un projet *open-source* qui connaît une popularité croissante dans le monde Java pour la construction d'applications 3-tiers. Son modèle de composants a été conçu sur des principes simples visant à fournir une alternative à la complexité de développement avec les traditionnels EJB [74]. Il est issu d'un ensemble de bonnes pratiques et de *Design Patterns* publiés par Rob Johnson en 2002 [72]. L'efficacité du modèle repose sur l'utilisation de simples objets Java, baptisés POJO (*Plain Old Java Object*).

Spring considère ainsi deux types de composants :

- les composants singletons : un composant singleton est associé à un nom unique et ses clients partagent une même instance d'objet. Ce type est particulièrement adapté pour des composants dont les services n'ont pas besoin de conserver un état pour chaque client.
- les composants prototypes : pour ces composants chaque appel provoque la création d'une nouvelle instance d'objet. Chaque client communique alors avec sa propre instance, ce qui permet d'éliminer des problèmes de concurrence au sein du fonctionnement du composant.

Pour spécifier les dépendances entre les composants, le modèle préconise une architecture basée sur des interfaces plutôt que sur des classes. Ainsi un composant ne dépend plus que d'une interface et non d'une implémentation ce qui limite le couplage des composants. La figure 1.5 montre l'architecture détaillée de l'assemblage d'un composant `Weather Center` avec un composant `Wind Sensor` à l'aide de l'interface `Wind Sensor ProvInt` . Le composant `Wind Sensor` peut ainsi être plus facilement remplacé par un autre composant implémentant la même interface `Wind Sensor ProvInt` sans intervenir sur l'implémentation du composant `Weather Center`. Cela est particulièrement intéressant pour effectuer des tests unitaires sur le composant `Weather Center` puisqu'il n'est pas nécessaire d'avoir développé les fonctionnalités du composant `Wind Sensor`, un simple composant implémentant l'interface `Wind Sensor ProvInt` suffit à réaliser un bouchon de test.

Spring n'invente rien de nouveau ici mais emploie des patrons de conception éprouvés et profite des constructions de base du langage Java, en l'occurrence les interfaces, pour ne pas alourdir son modèle. De même, le modèle se réapproprie les propriétés des JavaBeans et les utilise pour rendre les composants configurables et permettre de résoudre leurs dépendances envers d'autres composants ou ressources. C'est ce qu'on appelle l'« injection des dépendances » [91]. Dans ce cas, les composants n'utilisent pas dans leur implémentation une API particulière pour retrouver les services qu'ils requièrent, comme cela peut se faire avec les EJB par le mécanisme de nommage JNDI (*lookup*). Au contraire, les composants Spring déclarent leurs dépendances à travers des propriétés JavaBeans. Une entité extérieure, généralement l'infrastructure, a alors la responsabilité d'assigner les ressources aux composants. Les listings 1.5 et 1.6 montrent ces deux différentes approches. Dans la première, on voit clairement ligne 4 que le code du composant

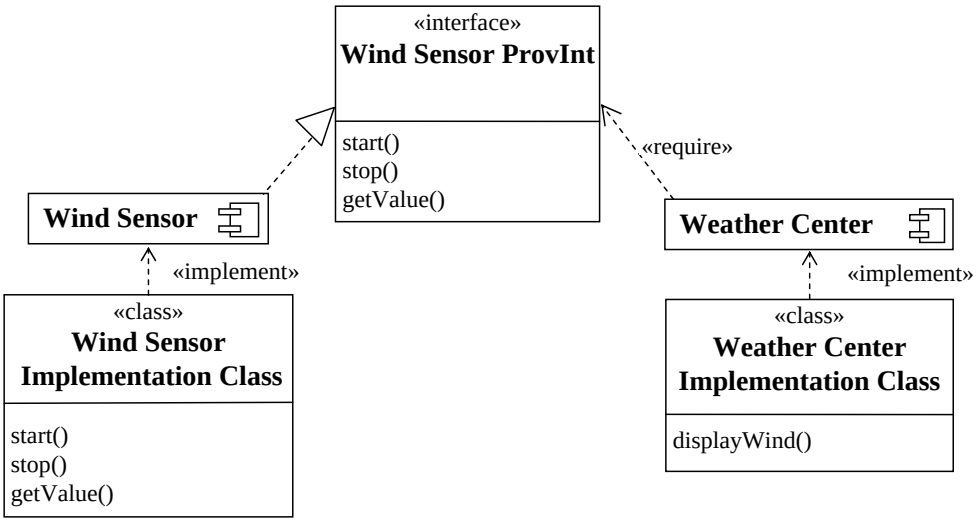


FIG. 1.5 – Architecture détaillée de l'assemblage d'un composant `Weather Center` avec un composant `Wind Sensor`

est directement lié à son environnement par l'appel `ctx.lookup()` déclenchant la recherche de la ressource nommée explicitement `"java :comp/env/resourceName"`. La seconde approche n'impose pas qu'une telle hypothèse sur le nom de la ressource soit directement implémentée dans le composant. L'assemblage, qui est initié en dehors du composant, consiste alors à transmettre au composant via la méthode `setResource()` la référence d'objet correspondant à la ressource.

```
1     private Resource resource;
2     // JNDI lookup
3     private void initResource(Context ctx) throws
         NamingException {
4         this.resource = (Resource) ctx.lookup("java:comp/env/
             resourceName");
5     }
```

LIST. 1.5 – Recherche d'une dépendance avec JNDI

```
1     private Resource resource;
2     // Setter Injection
3     public void setResource(Resource resource) {
4         this.resource = resource;
5     }
```

LIST. 1.6 – Injection d'une dépendance dans Spring

Fractal, un modèle hiérarchique

Le modèle de composant Fractal [23] a été défini par France Télécom R&D et l'INRIA dans le cadre du consortium ObjectWeb pour le *middleware open source*. Sa spécification n'est pas liée à un langage de programmation ou une technologie particulière mais son implémentation de référence, Julia, est écrite en Java.

Fractal a été développé conjointement avec son langage de description d'architecture, Fractal ADL. Son modèle de composants intègre ainsi de façon explicite les concepts des ADL comme le montre la figure 1.6 :

- les composants possèdent une membrane qui délimite clairement leur contenu de leur environnement extérieur afin de structurer l'application. Cette membrane dispose d'interfaces externes (resp. internes) permettant la communication des composants avec l'extérieur (resp. au sein de leur contenu). Leur contenu consiste en un ensemble fini de sous-composants. Le modèle est donc hiérarchique, ou récursif, ce qui est un principe caractéristique de la plupart des ADL.
- les connecteurs sont assimilés à des liens d'assemblage qui relient les composants par leurs interfaces. Les interfaces peuvent être soit des interfaces clientes, c'est-à-dire des interfaces exposant les services fournis par le composant, soit des interfaces serveurs, c'est-à-dire des interfaces qui correspondent aux services requis par le composant. Dans le cas de communications complexes, c'est-à-dire impliquant par exemple des besoins de distribution ou des modes de communication particuliers (synchrone, asynchrone, ...), les connecteurs peuvent être réalisés par des composants de liaison, qui implémentent donc un type particulier de communication entre composants.
- une configuration correspond à l'assemblage même des composants sélectionnés pour constituer l'application.

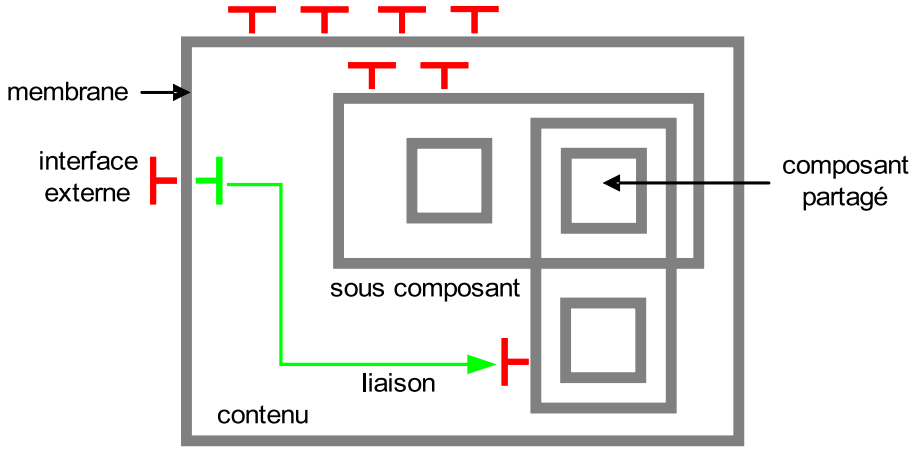


FIG. 1.6 – Exemple d'un composant Fractal (extrait de [36])

Le caractère hiérarchique du modèle Fractal entraîne la distinction entre trois types de composants :

- les composants composites sont des composants qui contiennent des sous-composants. Ils permettent d'obtenir une vue uniforme de l'application à différents niveaux d'abstraction.
- les composants primitifs sont des composants dont le contenu n'est pas exprimé en termes d'assemblage de composants. Ces composants peuvent servir à encapsuler des composants qui n'appartiennent pas au modèle de composant Fractal ou être construits simplement à base d'objets.
- les composants partagés sont des composants qui sont sous-composants de plusieurs composants composites. Ils permettent de modéliser notamment le partage des ressources entre composants.

En fusionnant les concepts des ADL, Fractal se démarque des modèles de composants que nous avons étudiés précédemment. Son principal avantage est la prise en compte de l'architecture des composants tout au long du cycle de vie du logiciel, notamment grâce aux différents types de composants qu'il définit.

1.1.3 Infrastructure d'accueil de composants

En complément de son modèle de composants à proprement parler, une technologie à composants fournit une infrastructure d'accueil qui implémente des services techniques pour appliquer et supporter son modèle de composants.

> *« A component framework is an implementation of services that support or enforce a component model. »* [8]

La raison d'être de l'infrastructure est de simplifier le travail du développeur en permettant la séparation des préoccupations entre code fonctionnel et code non-fonctionnel. Le développeur de composant peut se concentrer sur la logique applicative de ses composants qui seront contrôlés par l'infrastructure pour leur fournir les services techniques dont ils ont besoin. Dans l'optique de l'administration des composants, nous nous intéresserons en particulier au service de gestion du cycle de vie des composants.

Contrôleurs *vs.* composants métier

Dans les modèles de composants aussi bien industriels qu'académiques, les composants sont principalement considérés comme des entités métier. Ils sont l'incarnation de la description par le concepteur de la logique applicative du système en concepts clés clairement identifiés. Ces composants sont nommés dans la littérature « composants métier » [27] — *business components* en anglais.

Les composants métier ne sont qu'une facette de l'application : ils n'encapsulent que le code fonctionnel. Le code non-fonctionnel, complémentaire et nécessaire au fonctionnement de l'ensemble, est fourni par l'infrastructure dans laquelle sont déployés les composants. L'interaction entre les parties fonctionnelles et non-fonctionnelles est gérée au sein de l'infrastructure par des contrôleurs — également appelés conteneurs ou composants système [122] — responsables du contrôle des composants métier.

Les différents contrôles que peuvent opérer les contrôleurs sur les composants sont propres aux services que proposent l'infrastructure. Les technologies industrielles telles que Java EE, CCM, et .NET, proposent de nombreux services de haut niveau qui font la valeur ajoutée de ces solutions. Parmi ces services on trouve par exemple un service de gestion du cycle de vie des composants, un bus de communication pour gérer la distribution, un service de transaction pour gérer la fiabilité des échanges, un service de persistance pour la mise en cache des composants à état, un service de notification pour gérer les communications événementielles.

Les facilités proposées par ces technologies ont cependant des contreparties. Les difficultés de mise en œuvre technique dues à la complexité des API de services proposées entraînent une forte dépendance entre les composants et leurs contrôleurs. Les composants sont alors étroitement liés à la technologie ce qui restreint leurs possibilités de réutilisation. Ces services nuisent également à la souplesse de l'application car les mécanismes sous-jacents sont de trop haut niveau pour être exploités par des composants de faible granularité. Cela conduit à développer des composants monolithiques complexes et réduit le champ des possibilités architecturales.

Pour pallier ces problèmes, les approches à modèles hiérarchiques proposent de gérer les composants à différents niveaux de granularité. Parmi ces initiatives, nous avons vu précédemment que Fractal propose une catégorie particulière de composants, appelés « composants composites » (cf. 1.1.2 p. 18). Les composants composites se distinguent par le fait qu'ils permettent de gérer leur contenu sous la forme d'une interconnexion de sous-composants, eux-mêmes pouvant être composites ou non. Fractal se limite à quelques services de bas niveau tels que l'introspection, la gestion des cycles de vie, la gestion des liaisons extérieures ou encore la gestion des attributs pour configurer les composants. Néanmoins, les mécanismes d'extension du modèle offrent la possibilité de créer de nouveaux contrôleurs, spécialement adaptés aux besoins du domaine d'application considéré.

Dans un même esprit de simplification des contrôleurs, les récentes approches dites à « conteneurs légers », telles que Spring (cf. 1.1.2 p. 16), Excalibur [5] ou PicoContainer [124], tirent les enseignements des problèmes inhérents aux solutions industrielles. Ces infrastructures légères se concentrent principalement sur la mise en œuvre du patron de conception « injection de dépendances » pour réaliser l'inversion de contrôle [91]. Il s'agit de supprimer les dépendances du composant envers une API spécifique en rendant le contrôleur totalement responsable de la recherche et de la fourniture des ressources que le composant requiert. Ainsi, ces infrastructures peuvent être employées pour organiser en sous-composants les composants monolithiques que les modèles de composants industriels classiques amènent souvent à concevoir [73]. Les différents niveaux de contrôles sont alors obtenus par composition d'infrastructures.

Un service particulier : la gestion du cycle de vie

Au niveau de l'infrastructure, le composant est perçu avant tout comme une unité de déploiement. L'approche classique pour la gestion des activités de déploiement est de s'appuyer sur le cycle de vie des composants. Ce cycle de vie est généralement représenté par un diagramme d'état qui modélise les différents étapes de fonctionnement du composant. Même si elles varient sensiblement d'un type de composants à l'autre ou d'un modèle de composants à l'autre, les phases essentielles du cycle de vie d'un composant peuvent être résumées par le schéma 1.7.

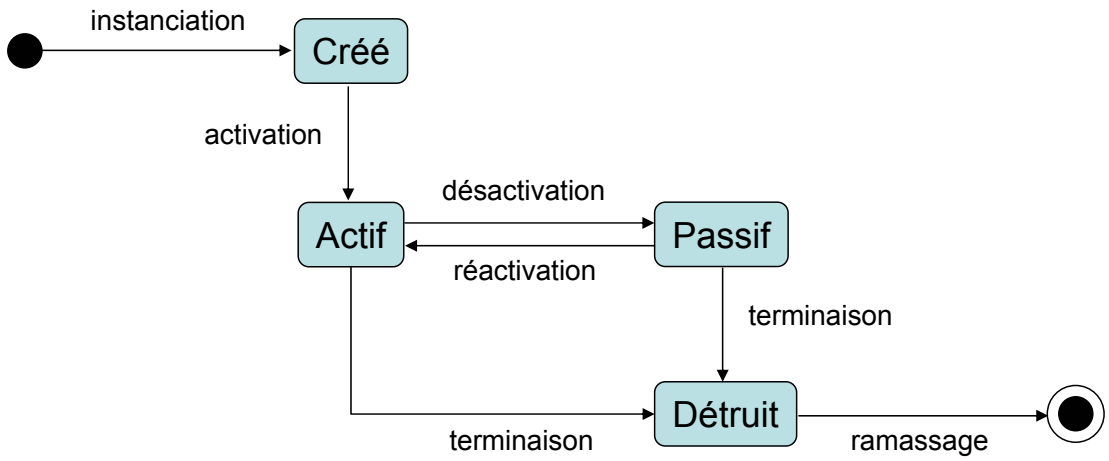


Fig. 1.7 – Cycle de vie d'un composant (adapté de [122, p. 276])

Lors de l'instanciation, une instance du composant est créée dans l'infrastructure pour y être placée sous l'autorité de son contrôleur. Les aspects métier et non-fonctionnels du composants sont alors recomposés. Après les configurations nécessaires, le composant peut être activé par son contrôleur pour exécuter les fonctionnalités qu'il implémente. Au cours de son exécution, il peut s'avérer nécessaire de stopper l'exécution du composant afin de procéder à des tâches auxquelles son évolution peut contraindre. Il peut s'agir par exemple d'une reconfiguration de l'application mettant en jeu des composants en relation avec le composant, ou encore pour permettre la persistance du composant.

Grâce au cycle de vie, on peut exprimer des contraintes sur les dépendances de vie des composants : par exemple, l'activation d'un composant peut entraîner l'activation automatique des composants dont il dépend ou au contraire la destruction d'un composant peut résulter de la destruction d'un autre composant. Il est ainsi possible de décrire des contraintes de cohérence sur l'état global d'une application.

Cependant, cet état ne traduit pas pour autant les réelles évolutions opérationnelles des composants. Il ne correspond qu'à différentes phases du processus de déploiement n'ayant pas forcément de liaison directe avec la logique métier des composants. Il peut se produire par exemple une défaillance interne dans un composant et que celui-ci soit toujours considéré comme étant « actif ». De plus, l'infrastructure impose sur le développeur l'implémentation d'interfaces

spécifiques fournissant les *callbacks* nécessaires à la gestion uniforme des composants. Il n'est pas toujours aisé pour le développeur de corréler le cycle de vie imposé à ses composants et leur comportement métier. Les technologies n'offrent pas non plus de facilités pour le faire et le développeur doit donc considérer lui-même les répercutions de ces *callbacks* sur l'exécution de ses composants.

Dans certains cas, il est malgré tout possible d'étendre le cycle de vie proposé par l'infrastructure. Fractal et Excalibur notamment proposent ce type d'extensions en offrant la possibilité de définir de nouvelles interfaces de contrôle. Leur mise en œuvre reste toutefois à la charge du développeur qui ne dispose toujours pas d'outils ni de méthodes rationnelles pour appréhender cette tâche de façon systématique et cohérente.

1.2 Ingénierie des composants logiciels dans les systèmes sans fil

La prolifération des appareils électroniques sans fil que l'on connaît aujourd'hui est le fruit des récentes avancées technologiques dans la miniaturisation des systèmes informatiques et dans le domaine des communications sans fil. L'amélioration des performances de ces systèmes allant de pair avec la complexification de leur logiciel, le développement d'applications sans fil impose dorénavant le recours aux méthodes du génie logiciel et en particulier au paradigme des composants logiciels.

Dans cette section, nous allons nous intéresser tout d'abord aux spécificités des systèmes sans fil. Puis nous présenterons un aperçu des différentes solutions à base de composants pour ces systèmes.

1.2.1 Caractéristiques des systèmes sans fil

L'entrée dans le monde du sans fil a fondamentalement modifié l'aspect et l'architecture des dispositifs communicants et de leurs réseaux. Pour en apprécier les conséquences sur le logiciel, il est important de présenter les caractéristiques des ces nouveaux appareils et leurs moyens de communication.

Des appareils hétérogènes

Les systèmes sans fil ont pour particularité commune évidente de recevoir et de transmettre des informations par radio grâce un dispositif ne faisant pas intervenir de connexion physique comme une prise ou un câble. Mais à quoi ressemble réellement un système sans fil ? C'est une question à laquelle il est difficile de répondre simplement, tant les solutions proposées diffèrent de par leur architecture, leur forme, leur utilisation, leur capacité de calcul, de mémoire, etc. La figure 1.8 présente un aperçu d'une partie des différents appareils sans fil qu'il existe à ce jour.

FIG. 1.8 – Hétérogénéité des appareils sans fil

Les téléphones mobiles sont les représentants emblématiques des systèmes sans fil. Ils font partie des appareils de poche [160], catégorie dans laquelle on trouve également les organiseurs, ou PDA (*Personal Digital Assistants*), qui sont des assistants personnels électroniques et les *pagers* qui sont des accessoires de radiomessagerie. Il est de plus en plus difficile de différencier ces trois types d'appareils de poche. La tendance s'oriente vers une sorte de « couteau suisse électronique » aux applications les plus variées, véritable hybride entre un téléphone mobile, un organiseur, un lecteur multimédia, une console de jeux et même un appareil photo.

Les capacités de communication sans fil de ces appareils accélèrent encore leur mutation. En effet, ils intègrent maintenant toutes les applications d'Internet et disposent entre autres de navigateurs Web, de clients de messagerie électronique, de lecteurs de flux multimédia, de logiciels de voix sur IP. Une conséquence importante de l'accès aux réseaux et à Internet est que ces systèmes sont maintenant des systèmes ouverts. Ils ont la capacité d'être modifiés à tout moment par le téléchargement de nouvelles applications et fonctionnalités à travers les réseaux sans fil.

La création de ces réseaux sans fil rend également possible l'interaction de ces différents appareils de poche entre eux, mais aussi avec des systèmes sans fil plus volumineux dont les performances sont plus proches de celles des ordinateurs de bureau classiques. Par exemple, un PDA peut synchroniser les données de ses applications avec celles d'un ordinateur portable, un boîtier de télévision peut projeter sur le téléviseur auquel il est relié les contenus multimédia des

appareils sans fil environnants, un téléphone mobile peut télécharger les photos qu'il contient directement sur une imprimante pour les imprimer. Ce type de réseaux trouve également de nombreuses applications en domotique mais aussi en automobile, que ce soit pour relier un système de chauffage à son boîtier de contrôle ou pour mettre à jour les cartes d'un système de navigation embarqué par exemple.

Si l'hétérogénéité des systèmes sans fil multiplie les possibilités d'applications, elle diminue en revanche la portabilité du logiciel et complexifie sa réutilisation. En effet, la capacité mémoire, la puissance de calcul, le type de connexion, les caractéristiques physiques des interfaces homme-machine sont autant de facteurs dont il faut tenir compte lors du développement du logiciel.

La plate-forme J2ME (*Java 2 Platform, Micro Edition*), récemment rebaptisée Java ME, tente de minimiser la contrainte d'hétérogénéité des environnement d'exécution. Elle s'appuie sur le langage Java qui s'exécute sur une machine virtuelle, la JVM (*Java Virtual Machine*), restreignant ainsi le problème de portabilité à la portabilité de la machine virtuelle. Cependant, Java est bien plus qu'un simple langage de programmation et sa machine virtuelle est composée d'une collection très complète d'API. Bien que ces API fassent la force de Java pour les systèmes classiques en constituant une large base d'outils de développement directement réutilisables, elles nécessitent néanmoins des ressources non négligeables pour des systèmes fortement contraints comme peuvent l'être les systèmes sans fil. Comme le montre la figure 1.9, l'architecture de Java ME a ainsi été réduite en taille par rapport à l'édition standard mais elle a en même temps été repensée pour permettre modularité et adaptation. Java ME se compose de deux configurations, CLDC (*Connected, Limited Device Configuration*) [146] et CDC (*Connected Device Configuration*) [149], qui déterminent une plate-forme minimale pour deux catégories d'appareils ayant des performances similaires. CLDC s'adresse en particulier aux appareils de poche tandis que CDC s'adresse aux appareils plus performants tels que les boîtiers de télévision. Les API proposées sont des sous-ensembles des API de la plate-forme standard auxquelles sont rajoutées des librairies spécifiques à ces deux catégories (cf. figure 1.10). Au-dessus de ces configurations, Java ME propose différents profils correspondants plus aux besoins des appareils d'un même domaine d'application comme la téléphonie, l'automobile ou la télévision. Même si elle ne résout pas tous les problèmes, Java ME est actuellement la plate-forme la plus utilisée sur les systèmes sans fil et fournit véritablement un standard pour le développement du logiciel sans fil [162].

Communications sans fil

De nombreuses technologies existent pour assurer les communications dans un réseau de terminaux « sans fil ». On les distingue selon deux catégories de réseaux : les réseaux de mobiles et les réseaux sans fil [1].

Les réseaux de mobiles sont issus du domaine de la téléphonie cellulaire. Ce sont des réseaux globaux implantés par des opérateurs de téléphonie dont la couverture s'étend quasiment sur l'ensemble du territoire. La première génération, dite « 1G », était analogique et ne transmettait que la voix, ce qui est le but principal de ces réseaux. Avec l'augmentation des capacités de

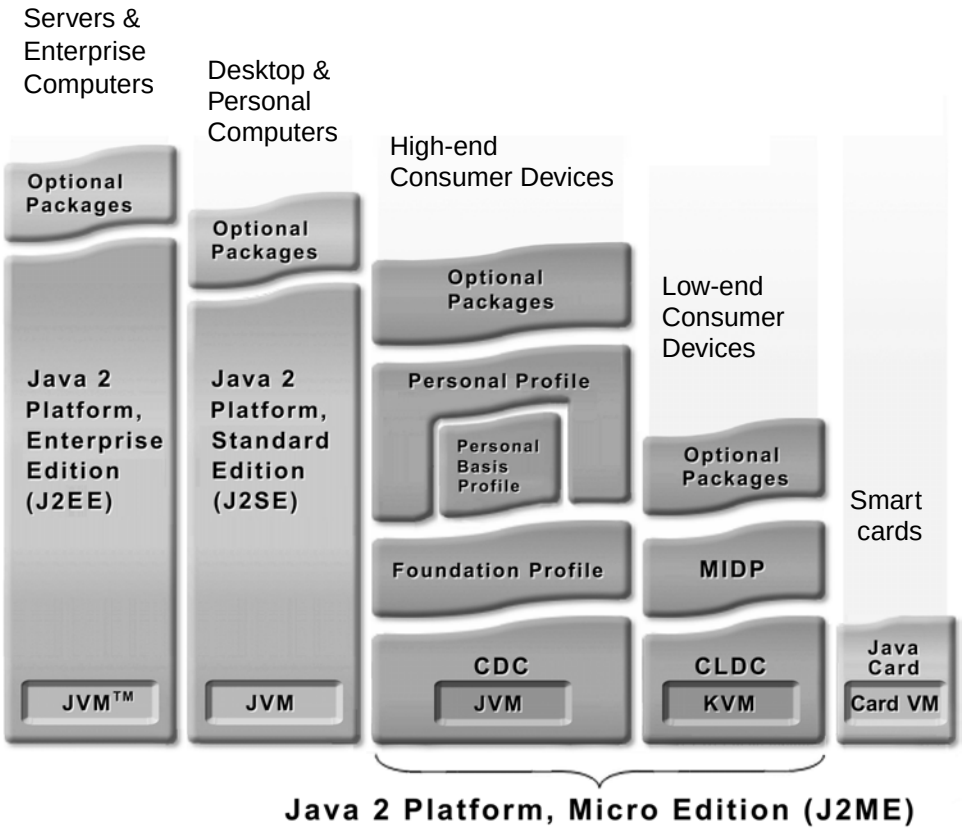


FIG. 1.9 – Java et ses différents domaines d'application [126, p. 8]

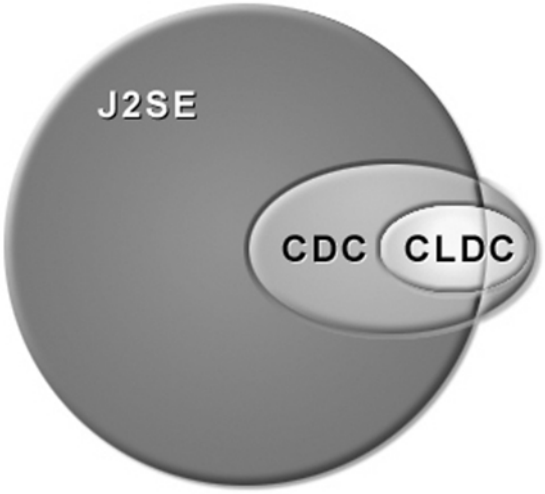


FIG. 1.10 – Correspondance entre les configurations J2ME et J2SE [126, p. 16]

calcul des appareils — « autrefois connus sous le nom de téléphones cellulaires » [53] — les nouvelles générations de réseaux mobiles transmettent également des données numériques. Quatre générations se sont depuis succédées, le passage d'une génération à une autre se caractérisant par une augmentation des débits possibles, de 9,6 Kbit/s pour la « 2G » à 2 Mbit/s théoriques pour la « 4G ». Le tableau 1.1 dénote cette évolution à travers les spécificités techniques des technologies représentatives de chaque génération.

Technologie	**Portée**	**Débit**	**Fréquences**
2G (GSM)	30 km	9,6 Kbit/s	800 / 900 / 1800 / 1900 MHz
3G (GPRS)	30 km	115 Kbit/s	800 / 900 / 1800 / 1900 MHz
4G (UMTS)	50 km	144 Kbit/s - 2 Mbit/s	1900 MHz - 2170 MHz

TAB. 1.1 – Caractéristiques techniques des réseaux de mobiles

Par rapport aux réseaux de mobiles, les réseaux sans fil n'appliquent pas de traitement particulier au transport de la voix, toute la bande passante est dédiée à la transmission de données — puissent-elles être effectivement des données audio. Comme le montre le tableau 1.2, toutes les technologies sans fil atteignent les 2 Mbit/s maximum théorique des réseaux de mobiles. Le Wifi [68] et le WiMax [69] surclassent même nettement ce débit avec respectivement 54 Mbit/s et 70 Mbit/s.

Ces technologies sont également moins consommatrices en énergie. Elles consomment en moyenne entre 1 mW et 100 mW contre 1 W pour la téléphonie mobile. Cela est dû au fait que les technologies sans fil ont été conçues pour des réseaux couvrant des zones géographiquement plus limitées.

L'IrDA [83] et Bluetooth [18] sont utilisés dans des réseaux de taille très restreinte appelés réseaux personnels, ou PAN (*Personal Area Network*). L'IrDA, qui utilise les ondes infrarouges, impose même un degré d'alignement entre émetteurs et récepteurs pour assurer la transmission des données. Bluetooth au contraire n'impose pas cette contrainte. La couche physique d'un réseau Bluetooth est mise en œuvre par ondes radio, ce qui permet une plus grande mobilité. Ces technologies ayant les portées les plus faibles, ce sont aussi les plus économes en énergie avec une consommation de l'ordre du milliwatt.

La technologie Wifi utilise les mêmes fréquences que Bluetooth, soit 2,4 GHz, mais nécessite des puissances d'émission comprises entre 60 et 100 mW pour atteindre des portées suffisantes pour établir des réseaux locaux, ou LAN (*Local Area Network*).

L'arrivée du WiMax permet d'envisager des réseaux sans fil à l'échelle d'une ville, ou MAN (*Metropolitan Area Network*). La portée offerte par cette technologie vient directement concurrencer les technologies de téléphonie mobile avec de plus une amélioration des débits d'un ordre de grandeur. Cela permettra à terme d'établir des réseaux sans fil ayant une couverture globale avec des débits de transmission de plus en plus proche des réseaux filaires.

Si les débits des réseaux filaires classiques, de l'ordre de 100 Mbit/s, commencent à être rattrapés, la qualité de service n'est pas encore la même. La variation des débits dus à l'atténuation

Technologie	Portée	Débit	Fréquences
IrDA	1 m	2,4 Kbit/s - 4 Mbit/s	790 nm - 1 mm
Bluetooth	1 m - 10 m	0,5 Mbit/s - 3 Mbit/s	2,4 GHz
Wifi (802.11)	25 m - 100 m	6,5 Mbit/s - 54 Mbit/s	2,4 GHz
WiMax (802.16)	10 km - 50 km	10 Mbit/s - 70 Mbit/s	2 GHz - 11 GHz

Tab. 1.2 – Caractéristiques techniques des réseaux sans fil

du signal radio causée par des obstacles ou par l'éloignement, les ruptures de connexions dues à des interférences ou à des déplacements hors de portée du réseau, la consommation d'énergie importante notamment en émission sont autant de paramètres critiques qu'il faut prendre en compte dans le développement du logiciel utilisant des connexions réseaux sans fil [90].

1.2.2 Modèles de composants spécifiques au sans fil

Il existe de nombreux modèles de composants permettant de développer des systèmes sans fil. La plupart sont directement adaptés des modèles classiques existants. Nous avons choisi d'en présenter quatre parmi les plus représentatifs. Les deux premiers modèles, Java ME et OSGi, définissent les deux standards les plus utilisés dans les systèmes sans fil. Ces deux initiatives sont issues du monde industriel mais sont la base de nombreux modèles académiques. Les deux modèles suivants sont caractéristiques des modèles académiques qui se positionnent sur des problématiques spécifiques des systèmes sans fil. Ainsi SATIN s'intéresse à la mobilité et FarGoDA aux déconnexions de ces systèmes.

Java ME : MIDlet et Xlet, des composants rudimentaires

Pour chacune de ses deux configurations [149, 146], Java ME définit un modèle de composants, les MIDlets et les Xlets. Les MIDlets font partie des profils MIDP (*Mobile Information Device Profile*) et PDAP (*Personal Digital Assistant Profile*) fonctionnant au dessus de la configuration CLDC qui s'adresse aux systèmes les plus contraints, comme les téléphones mobiles ou les PDA. Les Xlets sont disponibles dans les profils PBP (*Personal Basis Profile*) et PP (*Personal Profile*) utilisables notamment avec les boîtiers de télévision ou les PDA de dernière génération capables de supporter la configuration CDC.

Ces deux modèles de composants sont issus des applets, le tout premier modèle de composant du monde Java [154]. Les applets sont des applications téléchargeables et exécutables au sein des pages affichées par les navigateurs Web. Leur modèle d'exécution est conforme au cycle de vie classique des composants (cf. figure 1.7 p. 21). Il permet au navigateur d'exercer un contrôle sur l'application intégrée afin qu'elle ne perturbe pas sa propre logique applicative. L'applet peut par exemple être mise en pause ou même détruite lors de l'activation d'un hyperlien vers une cible externe, et redémarrée ou recréée au retour sur la page originale.

Les MIDlets et les Xlets reprennent un modèle d'exécution similaire. Le cycle de vie de ces composants est géré par un logiciel résidant sur le système sans fil, l'AMS (*Application Management System*). Cette gestion est cruciale pour les systèmes sans fil car ils possèdent des ressources limitées. Il faut donc penser à minimiser leur occupation et les partager entre les différentes applications. De plus, les systèmes sans fil sont des systèmes réactifs qui doivent faire face à des événements prioritaires impliquant l'exécution d'une application en faveur d'une autre. L'AMS, qui travaille en collaboration étroite avec le logiciel natif de l'appareil, est par exemple responsable lors d'un appel reçu sur un téléphone mobile de l'arrêt du composant en cours d'exécution et de sa reprise ultérieure.

Le modèle de composants proposés par les MIDlets est rudimentaire. La granularité des composants est l'application et il n'existe pas de mécanismes d'interface pour permettre aux composants de collaborer. Les composants de MIDP sont pensés comme unités de déploiement mais pas vraiment comme unités de composition. Leur composition se résume à gérer leur exécution au sein du système. Le modèle des Xlets est amélioré par rapport aux MIDlets par l'introduction de mécanismes de communication entre les composants du même système sans fil. Cependant ces mécanismes sont peu flexibles et lourds à mettre en œuvre car ils sont hérités des serveurs d'application qui les utilisent pour des communications distantes.

Ces modèles facilitent néanmoins le déploiement de nouvelles applications sur les systèmes sans fil qui ne disposent souvent que d'un système d'exploitation minimal. En effet, l'AMS qui contrôle ces applications permet également d'en télécharger de nouvelles via des communications sans fil, les composants jouant un rôle de conteneur pour ces applications.

OSGi, des composants de service

La plate-forme OSGi [119] fournit une infrastructure de services au dessus des environnements d'exécution Java mais cible en particulier Java ME. Le principe est différent des applications classiques puisque justement, OSGi ne définit pas de notion d'application à proprement parler [44]. L'infrastructure représente l'application principale. En constante exécution, elle permet d'ajouter ou de supprimer dynamiquement des services qui peuvent collaborer selon le paradigme de la programmation orientée services.

Les services sont encapsulés dans des composants appelés *bundles* et correspondent à un paquetage Java (fichier JAR). Comme dans Java ME, l'infrastructure contrôle la phase de déploiement des composants en gérant non seulement leur cycle de vie mais en permettant également leur installation à partir d'une source distante. La différence réside dans le fait que l'infrastructure n'a pas besoin d'être arrêtée pour installer ou désinstaller des composants, autorisant ainsi les services actifs à poursuivre leur exécution. Des services peuvent donc apparaître ou disparaître à tout moment. L'infrastructure est alors responsable de la notification des différents changements auxquels les composants doivent faire face.

La collaboration inter-composants s'effectue par des interfaces qui établissent des contrats de service bien définis. Les composants sont alors liés à un service et non directement à un

composant fournissant le service. Cela facilite la mise à jour et le remplacement des composants. OSGi permet également une autre forme de dynamicité par l'ajout de composants étendant les fonctionnalités de composants déjà activés. Cette liaison tardive entre composants rend possible le développement d'applications à base de *plugins*. C'est le cas par exemple d'Eclipse qui utilise OSGi pour la gestion de ses greffons.

SATIN, des composants mobiles

SATIN (*System Adaptation Targeting Integrated Networks*) [166] définit un modèle de composants adaptés aux applications ubiquitaires. Ce modèle fournit des mécanismes pour faciliter la construction de systèmes sans fil réactifs et adaptables à leur environnement. Implémenté sur Java ME CDC, il met en œuvre en particulier un service de mobilité du code et un service de publication/découverte pour les composants.

L'environnement considéré se compose d'un réseau de systèmes sans fil dans lesquels sont déployés les composants. Une infrastructure classique à contrôleur permet de gérer localement ces composants selon leur cycle de vie. Lorsqu'un nouveau système sans fil est introduit dans l'environnement, ses composants ont la capacité d'annoncer leur présence à tout le réseau en implémentant une interface spécifique. A l'inverse, les autres composants du réseaux sont notifiés de la présence d'un nouveau composant via leur infrastructure.

Suivant leurs différents besoins d'adaptation, les différents systèmes sans fil du réseau peuvent s'échanger dynamiquement des composants par des techniques de mobilité de code. SATIN définit pour cela les concepts d'entité et d'unité de mobilité logique, LME (*Logical Mobility Entity*) et LMU (*Logical Mobility Unit*). Les LMEs correspondent à des constructions du systèmes comme les composants, les classes, les instances, ou les types de données. Les LMUs servent à encapsuler les LMEs et à décrire leurs dépendances. Une fois qu'une LMU est composée, l'infrastructure permet sa sérialisation, son transfert vers un hôte distant, sa désérialisation et son déploiement.

FarGo-DA, des composants supportant les déconnexions

Les déconnexions survenant dans un système sans fil ne sont pas toutes imprévisibles et peuvent même être provoquées volontairement par l'utilisateur du système. C'est le cas par exemple lorsque l'utilisateur coupe les communications sans fil pour économiser les batteries de son appareil. Le modèle de composants FarGo-DA [159] a été défini pour supporter la gestion des déconnexions et des reconnexions. FarGo-DA fait l'hypothèse que le système sans fil est utilisé comme client léger d'une application distribuée. Lorsqu'une déconnexion est programmée, les mécanismes de l'infrastructure permettent de préserver un système minimal opérationnel en mode déconnecté.

Etant dans un contexte distribué, des composants du systèmes sans fil établissent des relations avec des composants distants. La sémantique des relations entre composants dans FarGo-DA est augmentée pour définir comment doit être gérée leur déconnexion et leur reconnexion. Plusieurs politiques de gestion sont prévues.

Concernant les déconnexions, la première utilise des techniques de mobilité de code pour cloner le composant distant, et transitivement les composants distants dont il dépend. Au contraire, on peut choisir de remplacer le composant distant par un composant local possédant la même interface mais avec une implémentation plus légère. Une autre politique consiste à traiter de façon asynchrone les invocations distantes ayant lieu en mode déconnecté. Les invocations sont mises en file d'attente et l'infrastructure se charge de les reprendre dès que la connexion est rétablie.

En mode déconnecté, les états des composants clonés ou remplacés peuvent évoluer indépendamment. A la reconnexion, la cohérence des états de ces composants peut être rétablie par différentes méthodes. L'infrastructure peut écraser l'état du composant cloné ou remplacé par celui du composant original ou inversement selon la politique choisie (priorité à l'original, dernier modifié, ...). Dans des cas conflictuels, l'application peut elle-même fournir sa propre méthode de résolution de la cohérence des états.

1.3 Du besoin d'administration pour les composants sans fil

Le caractère immatériel et ubiquiste du médium physique qu'utilisent les systèmes sans fil dans leurs communications, leur confère une propension à l'interaction. Ils sont facilement connectés aux systèmes environnants ce qui leur permet d'exploiter des fonctionnalités pour lesquelles ils ne sont pas conçus eux-mêmes. Si les systèmes sans fil étaient d'abord dédiés à des tâches particulières et figées, l'introduction du paradigme des composants dans le développement logiciel de ces systèmes leur apporte flexibilité, dynamisme et ouverture. Les modèles de composants que nous avons étudiés précédemment montrent bien cette évolution. Java ME autorise le téléchargement de nouvelles applications permettant d'étendre les fonctionnalités du système et OSGi permet même l'intégration à l'exécution de nouveaux composants, tandis que SATIN va encore plus loin en donnant la possibilité aux systèmes de découvrir et de s'échanger leurs composants en fonction des modifications de l'environnement.

L'ouverture de ces systèmes place le logiciel dans un contexte dans lequel interviennent de nombreux changements qui peuvent même ne pas avoir été anticipés. Baresi *et al.* ont récemment identifié les défis de ces nouveaux paramètres et soulignent en particulier la nécessité de pouvoir observer et contrôler le comportement du logiciel en exécution :

> *« A highly dynamic and open system necessitates runtime monitoring to watch for situations that might require suitable reactions to assure the desired level of global quality. [...] As a result of monitoring, it should be possible to handle deviations from expected behaviors and plan for a reconfiguration. »* [14]

Plusieurs études abondent dans ce sens et montrent que ce manque est particulièrement important pour les composants logiciels embarqués dans ce type de systèmes [38, 106, 152]. En effet, les composants doivent faire face à des situations d'instabilité particulières qui sont

liées au sans fil, comme par exemple les déconnexions dans le modèle de composants FarGo-DA. L'administration des composants logiciels est d'une manière générale connue pour être un problème crucial et difficile à résoudre [157] mais les caractéristiques du sans fil font que cette tâche est encore complexifiée et doit donc être traitée spécifiquement.

Chapitre 2

Administration d'applications logicielles

Sommaire

2.1 Concepts et principes 34
2.1.1 Définition générale de l'administration 34
2.1.2 Le modèle de gestion OSI 35
2.1.3 Activités de l'administration 37
2.2 Approches classiques 40
2.2.1 Approches centralisées 41
2.2.2 Approches hiérarchiques 42
2.2.3 Approches par politique 43
2.2.4 Approches autogérées 44
2.3 Administration des composants logiciels 46
2.3.1 Approches basées sur l'infrastructure 46
2.3.2 Approches basées sur l'architecture 54
2.3.3 Approches basées sur le comportement 58
2.4 Solutions d'administration des composants logiciels sans fil 62
2.4.1 OMA Device Management 62
2.4.2 JSR 232 & JSR 233 66
2.4.3 Dynamically Programmable and Reconfigurable Software 68
2.5 Synthèse 71

Les défaillances des systèmes informatiques peuvent avoir des conséquences considérables. Dans les domaines des systèmes, des réseaux ou des télécommunications, l'administration[4] est une activité primordiale qui permet de gérer le fonctionnement de ces systèmes afin de détecter au plus vite les problèmes qui peuvent survenir et de les corriger. A l'opposé, cette activité est souvent peu considérée pour les applications logicielles elles-mêmes. Pourtant, la complexité de

[4]De l'anglais *management*, le terme est également traduit par supervision ou gestion.

ces systèmes est telle qu'une tolérance totale aux fautes n'est pas possible. En effet, malgré les efforts de l'ingénierie logicielle dans la qualité et la sûreté de fonctionnement du logiciel, il n'est pas rare que des bogues apparaissent inopinément. Il y a donc un besoin crucial d'administration.

Après une introduction aux concepts et aux problématiques de l'administration, ce chapitre présente les approches existantes, en particulier les approches dédiées aux composants logiciels et plus spécifiquement encore celles qui traitent les composants dans les systèmes sans fil.

2.1 Concepts et principes

2.1.1 Définition générale de l'administration

En informatique il existe plusieurs types d'administration. Par exemple, l'administration réseau concerne les équipements matériels qui constituent un réseau et le réseau lui-même vu comme une entité abstraite. Dans le premier cas, il s'agit de superviser l'état des routeurs, des switches, des hubs et des autres appareils formant l'infrastructure du réseau. Dans le second cas, il s'agit de maintenir la qualité de service délivrée par le réseau. Un autre exemple d'administration est l'administration système qui gère l'exécution d'un système sur un ordinateur à travers l'utilisation de ses ressources (comme le processeur, la mémoire, le disque), des processus, du système de fichier, etc. Un dernier exemple, plus proche de notre propos, est l'administration d'applications. Ici, l'objectif est de détecter et de réparer les problèmes qui peuvent survenir au cours du cycle de vie de ces applications, notamment lors de l'exécution.

Bien que les systèmes gérés diffèrent, les limites entre les différents types d'administration sont assez floues et on parle même d'administration intégrée couvrant tous les types d'administration [93]. Quoi qu'il en soit, ces solutions partagent le même but général : s'assurer du bon fonctionnement d'un système. Pour cela les mêmes principes de base sont appliqués. L'activité d'administration peut ainsi être exprimée par l'équation suivante :

« *Management = Monitoring + Controlling* » [143]

Cette équation fait apparaître clairement les deux activités essentielles de l'activité d'administration : la surveillance (observation) et le contrôle (action) [49]. Le schéma de la figure 2.1 décrit l'architecture fondamentale de l'administration selon ces deux activités. On distingue ainsi deux entités principales : le système d'administration et la ressource administrée.

Le système d'administration est responsable de la gestion de la ressource administrée. Il interagit avec la ressource à travers des capteurs pour l'observer et des activateurs, ou actionneurs, pour la contrôler. Les systèmes d'administration possèdent différents degrés d'autonomie. Certains ne consistent seulement qu'en une console permettant à un administrateur humain de superviser les ressources, souvent à distance. D'autres plus évolués, peuvent décider eux-mêmes des actions à opérer sur les ressources en fonction des données qu'ils observent.

La ressource administrée a la responsabilité de fournir les capteurs et les activateurs nécessaires à sa gestion par le système d'administration : c'est ce qu'on appelle l'instrumentation.

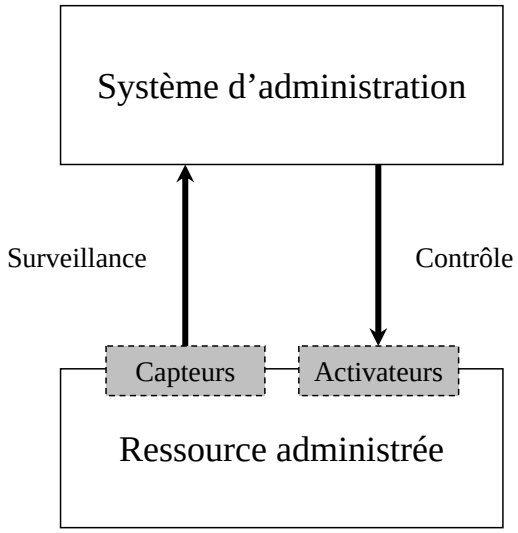


FIG. 2.1 – Architecture fondamentale de l'administration

Selon les fonctionnalités recherchées et les caractéristiques de la ressource à administrer, l'instrumentation peut être plus ou moins intrusive. Par exemple, pour une ressource possédant un fichier de configuration externe, la reconfiguration ne nécessite que l'accès au fichier de configuration sans qu'aucune modification de la ressource soit nécessaire. Les capteurs et les activateurs nécessaires existent déjà. Au contraire, l'observation et la modification de l'état d'une ressource en fonctionnement n'est que rarement accessible de l'extérieur. Il faut donc fournir les interfaces d'administration nécessaires pour accéder aux données internes désirées.

2.1.2 Le modèle de gestion OSI

Malgré des difficultés d'intégration des approches d'administration [49], les architectures d'administration sont traditionnellement décomposées en quatre modèles [31, 86] : un modèle organisationnel, un modèle d'information, un modèle de communication et un modèle fonctionnel.

Le modèle organisationnel

Le modèle organisationnel définit les entités et les rôles de l'architecture d'administration ainsi que leur composition. L'évolution de l'organisation des approches d'administration a suivi l'évolution des systèmes eux-mêmes [161]. Ainsi, comme les premiers systèmes n'étaient pas interconnectés, les infrastructures de gestion adoptaient une organisation exclusivement centralisée. Puis avec l'apparition des systèmes distribués, des solutions hiérarchisées ont été proposées

notamment pour pallier l'augmentation des systèmes à administrer. Par exemple, la technologie SNMP (*Simple Network Management Protocol*) reposait sur une architecture centralisé dans ses premières versions (SNMPv1, SNMPv2). Sa troisième version (SNMPv3) ajouta le support d'une administration hiérarchisée. Aujourd'hui, on trouve également des approches par politique, à agents intelligents et autogérées. Les architectures classiques seront étudiés en détail en section 2.2.

Le modèle d'information

Le modèle d'information définit la structure de l'information d'administration suivant laquelle sont décrites les ressources gérées. De multiples modèles existent pour ce faire, comme par exemple :

- les MIBs (*Management Information Bases*) de l'IETF (*Internet Engineering Task Force*). Utilisées notamment par SNMP, les MIBS sont des structures arborescentes dont les nœds peuvent être référencés par un identificateur unique. Le langage de description utilisé est SMI (*Structure of Management Information*) [98] qui est dérivé de la syntaxe d'ASN.1 [34].
- les modules PIB (*Policy Information Bases*), également de l'IETF, sont adaptés à la gestion par politique et sont décrits avec une syntaxe proche de SMI appelée SPPI (*Structure of Policy Provisioning Information*) [97].
- les MIBs comme définis originellement par l'ISO (*International Organization for Standardization*) utilisent le métamodèle orienté objet GDMO (*Guidelines for the Definition of Managed Objects*) [32].
- les schémas CIM (*Common Information Model*) sont également orientés objet et peuvent être décrits sous deux formes : une forme graphique avec le langage de modèlisation UML et une forme textuelle avec le MOF[5] (*Managed Object Format*) [41].

Le modèle de communication

Le modèle de communication définit les opérations et le protocole de communication pour accéder au modèle d'information. SNMP [25] a été le protocole prédominant dans les réseaux TCP/IP notamment grâce à sa simplicité. Il s'adresse en particulier à l'administration de matériels réseaux, comme les routeurs par exemple, qui ne nécessitent pas de fonctionnalités d'administration complexes. La figure 2.2 présente le protocole SNMP.

Face à la simplicité de SNMP, l'exhaustivité de CMIP [30, 70] (*Common Management Information Protocol*) n'a trouvé sa place que dans les environnements où le modèle OSI était déjà implanté, c'est-à-dire principalement dans les réseaux de télécommunications.

Le DMTF (*Distributed Management Task Force*) a ensuite proposé d'utiliser les technologies de l'Internet pour le transport des communications entre gestionnaires et agents. Le DMTF a

[5] A ne pas confondre avec le *Meta Object Facility* qui est le métamodèle d'UML.

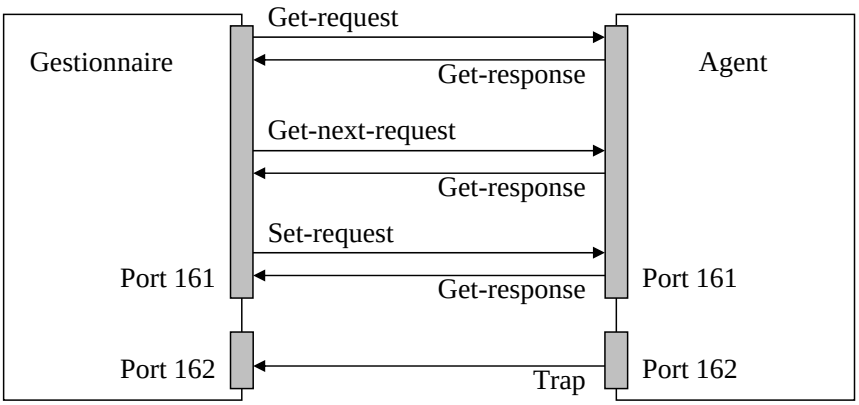


FIG. 2.2 – Le protocole SNMP

donc créé le WBEM (*Web-Based Enterprise Management*) qui s'appuie sur le protocole HTTP pour le transfert des opérations d'administration encodées en XML [42]. C'est donc naturellement que WBEM s'oriente aujourd'hui vers le protocole SOAP des services Web, SOAP étant également basé sur XML [161].

Le modèle fonctionnel

Le modèle fonctionnel définit les services d'administration génériques que l'on peut utiliser pour gérer chaque ressource. Les différentes solutions d'administration font apparaître des besoins différents en terme de fonctionnalités. Cela est principalement dû aux types de ressources qu'elles administrent.

2.1.3 Activités de l'administration

Dans le cadre d'un effort de standardisation de l'administration des réseaux, l'ISO a identifié cinq grandes catégories d'activités mises en jeu dans l'administration, connues sous l'acronyme FCAPS [29] :

- *Fault Management* (Gestion des fautes) : La gestion des fautes est l'activité prépondérante de l'administration [143]. Elle consiste essentiellement à surveiller les ressources administrées afin de détecter les anomalies de fonctionnement (les fautes) à l'instant où elles apparaissent et à opérer sur les ressources les actions correctives nécessaires afin de restaurer un état normal du système.
- *Configuration Management* (Gestion de la configuration) : Les ressources introduites dans un système ont souvent besoin d'être configurées pour fonctionner correctement dans leur nouvel environnement. La gestion de la configuration gère la configuration des différentes ressources afin de faciliter la configuration globale du système en fonction des besoins.

- *Accounting Management* (Gestion comptable) : Il s'agit de mesurer l'utilisation des ressources du système. Dans les réseaux de télécommunications, le but est notamment de calculer le coût des services fournis et de les facturer au client.
- *Performance Management* (Gestion de la performance) : Cette activité consiste à évaluer et à contrôler l'efficacité et la productivité des ressources gérées. Le temps de réponse et le débit de traitement font partie des principaux critères de performance pris en compte.
- *Security Management* (Gestion de la sécurité) : La gestion de la sécurité protège l'accès aux ressources et garantit leur intégrité grâce notamment à des mécanismes d'authentification et des techniques de chiffrement.

Bien que générales, les catégories proposées par l'ISO pour la gestion des réseaux ne correspondent pas de façon optimale à la problématique de gestion des applications [143]. Cette classification doit donc être repensée et affinée pour traiter spécifiquement les besoins de l'administration d'applications logicielles.

Kreger propose de se baser sur le cycle de vie d'une ressource administrée [85, 47, 86]. Elle fait le constat que presque toutes les ressources informatiques ont un cycle de vie similaire qu'il faut administrer. Ce cycle de vie est présenté en figure 2.3 et comprend les étapes suivantes :

1. *Distribute* (Distribuer) : Déployer les artefacts constituant la ressource dans le système cible.
2. *Install* (Installer) : Installer la ressource dans le système cible.
3. *Start* (Démarrer) : Initialiser la ressource afin qu'elle puisse être exécutée.
4. *Execute* (Exécuter) : Effectuer dynamiquement des opérations.
   (a) *Monitor* (Surveiller) : Vérifier périodiquement la ressource pour s'assurer de sa disponibilité et de son bon fonctionnement.
   (b) *Operate* (Opérer) : Invoquer durant l'exécution des fonctions ou des opérations qui n'affectent pas de façon permanente l'état de la ressource.
5. *Update* (Mettre à jour) : Mettre à jour la ressource.
   - *Configure* (Configurer) : Changer de façon permanente la configuration de la ressource à l'exécution ou à l'arrêt.
6. *Stop* (Arrêter) : Terminer l'exécution de la ressource.
7. *Maintain* (Maintenir) : Appliquer les modifications de code ou de configuration de la ressource périodiquement ou à la demande.
8. *Uninstall* (Désinstaller) : Retirer la ressource du système.

La quatrième étape, « exécuter », est l'étape primordiale du cycle de vie : c'est à ce moment que l'application fait ce pour quoi elle a été conçue. Elle se différencie des autres étapes par le fait que l'application est en cours d'exécution et doit le rester. Elle est donc une phase privilégiée pour l'administration et constitue ce que l'on appelle le *runtime management* (l'administration à l'exécution). On retrouve ainsi dans cette étape deux sous-activités correspondant aux deux

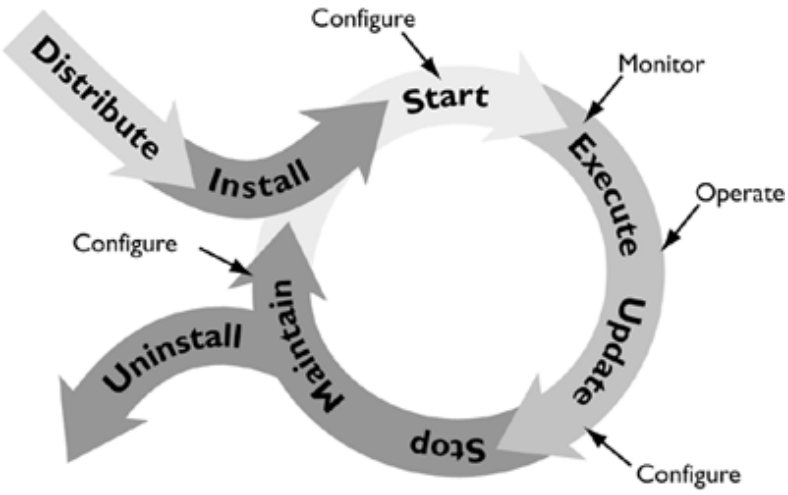


FIG. 2.3 – Cycle de vie d'une ressource administrée (extrait de [86, p. 16])

préceptes fondamentaux de l'administration surveiller et contrôler, appelés ici *monitor* et *operate*, dont le but est d'administrer le fonctionnement même de la ressource.

Comme dans les catégories de l'ISO, on retrouve dans le cycle de vie l'activité de configuration. La figure 2.3 montre qu'une application peut être configurée lors de différentes phases : au démarrage, pendant une mise à jour ou lors de la maintenance. Une mise à jour est souvent un processus lourd qui nécessite l'arrêt et le redémarrage de l'application administrée, mais dans certains cas la modification de configuration peut s'opérer à l'exécution, on parle alors de reconfiguration dynamique. C'est également une forme de *runtime management* mais ici on s'intéresse plus aux modes de fonctionnement de l'application plutôt qu'à l'évolution de l'exécution de ce fonctionnement.

Les autres étapes n'ont pour but que d'amener ou de ramener l'application dans son état d'exécution, excepté pour l'étape ultime « désinstaller » qui retire complètement la ressource administrée du système.

En résumé, nous pouvons dire qu'il s'agit toujours de surveiller et de contrôler l'application mais selon différents points de vue, c'est-à-dire vis-à-vis de son fonctionnement, de sa configuration ou de son cycle de vie. Cette analyse nous permet donc d'identifier trois types d'état à administrer sur une application :

- l'état de cycle de vie, qui concerne les différentes phases de l'artefact logiciel incarnant l'application, depuis sa création jusqu'à sa destruction en passant par son exécution,
- l'état de configuration, qui correspond aux paramètres de configuration qui déterminent quel fonctionnement l'application exécute,
- l'état de fonctionnement, qui traduit l'évolution de l'application au cours de son exécution pour réaliser la tâche ou la fonction pour laquelle elle a été conçue et configurée.

Dans cette thèse, nous nous intéresserons spécifiquement à l'administration de l'état de fonctionnement des applications et des composants logiciels qui les composent. Pour traiter cette question, il est important d'introduire à présent les approches classiques d'administration.

2.2 Approches classiques

La question de l'administration en informatique s'est fortement développée, particulièrement dans les réseaux, et de nombreuses approches ont été recensées [76, 94, 20, 142]. Les premiers efforts de recherche, de développement, de standardisation et de normalisation se sont accordés autour d'un modèle fédérateur qui sert encore de base au solutions actuelles [51] : le modèle « gestionnaire - agent ». Ce modèle affine l'architecture fondamentale que nous avons présentée précédemment (cf. figure 2.1).

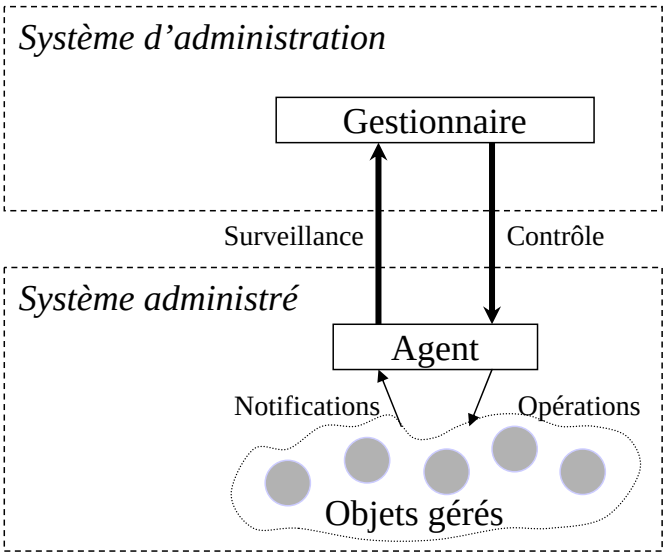


Fig. 2.4 – Le modèle gestionnaire - agent

Comme dans l'architecture fondamentale, on retrouve d'un côté le système d'administration avec le gestionnaire qui agit en tant qu'interface pour l'administrateur humain, et de l'autre le système administré comportant les ressources à gérer. Ce modèle introduit un nouvel acteur, l'agent, interposé entre le gestionnaire et les objets qu'il administre. L'agent agit comme délégué local du gestionnaire pour lequel il consulte ou modifie l'état des objets administrés via des notifications et des opérations. Le concept d'agent a principalement été introduit pour prendre en compte la distribution : l'agent est déployé sur le même nœud que les ressources à administrer et le gestionnaire opère sa surveillance et son contrôle à distance sur un autre nœud du réseau.

A partir de ce modèle général, Martin-Flatin propose dans sa thèse [94] une taxonomie simple

des systèmes d'administration selon les relations entre gestionnaires et agents. Cette taxonomie comprend quatre types de systèmes d'administration : centralisés, faiblement distribués, fortement distribués, et coopératifs.

Schönwälder fournit une formalisation de cette classification :

> « Soient g le nombre total des gestionnaires et a le nombre total d'agents.
> Soit $n = g + a$ le nombre total d'éléments du système d'administration.
> On peut distinguer quatre classes de système d'administration :
> - si $1 = g$ alors le système d'administration est centralisé,
> - si $1 < g \ll n$ alors le système d'administration est faiblement distribué,
> - si $1 \ll g < n$ alors le système d'administration est fortement distribué,
> - si $g \approx n$ alors le système d'administration est coopératif. » [136]

Dans cette section, nous présentons les approches de type centralisé, puis celles de type distribué, faiblement ou fortement, à travers les approches hiérarchiques et par politique, et enfin nous étudions comme approche de type coopératif les approches autogérées.

2.2.1 Approches centralisées

Les approches centralisées se caractérisent par le fait qu'un seul gestionnaire a la responsabilité de tout le traitement de la logique de l'administration. Les agents ne jouent le rôle que de *proxy* entre le gestionnaire et les ressources administrées. La figure 2.5 schématise l'architecture de ces approches.

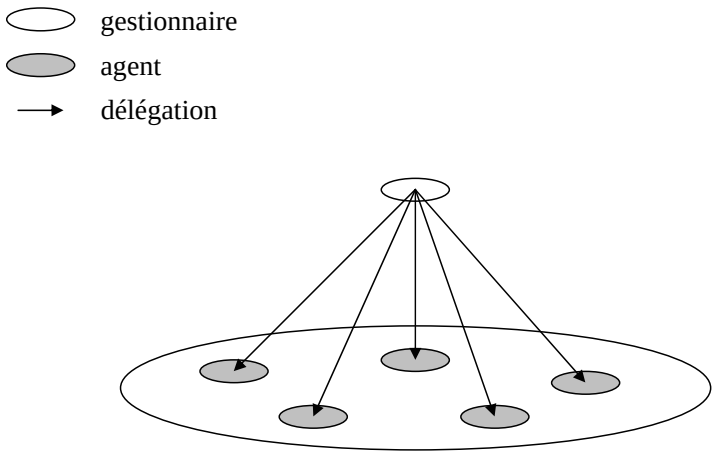


Fig. 2.5 – Approche d'administration centralisée

Le fait de centraliser l'administration est un avantage considérable pour les administrateurs puisqu'ils peuvent superviser toutes les ressources simultanément à partir d'une seule et même machine. Ils n'ont plus la contrainte d'effectuer les opérations d'administration individuellement

et directement sur chaque système. Les systèmes devant être administrés sont de plus déchargés d'une partie importante du poids de l'administration puisque cette activité est prise en charge par un gestionnaire qui peut être déployé sur un système dédié à l'administration.

L'approche centralisée a fait le succès des premières version de SNMP [25] qui fonctionnent bien pour l'administration de réseaux locaux de petite taille. Cette approche se retrouve également dans de nombreux équipements réseaux qui intègrent des serveurs HTTP pour offrir des solutions d'administration simples [96]. Si l'approche centralisée a prouvé son efficacité pour la gestion d'un nombre peu élevé de ressources, elle trouve par contre ses limites dans le passage à l'échelle [4] et ne peut faire face à l'augmentation du nombre des systèmes à gérer.

2.2.2 Approches hiérarchiques

L'interconnexion des réseaux a provoqué une explosion du nombre des systèmes à administrer, saturant dans le même temps les capacités des gestionnaires centralisés. Les approches hiérarchisées sont alors apparues pour repousser les limites des approches centralisées. Le principe des approches hiérarchiques est de partager le traitement de l'administration entre plusieurs gestionnaires [139, 58], responsables chacun d'un nombre rationnel de ressources à administrer. La figure 2.6 schématise l'architecture de ses approches.

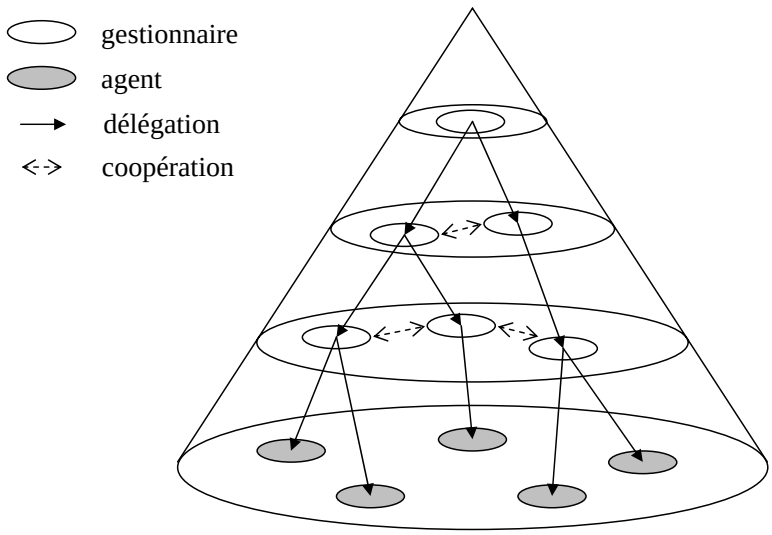


Fig. 2.6 – Approche d'administration hiérarchique

Le partage des responsabilités de l'activité d'administration est organisé en une hiérarchie de gestionnaires. Chaque gestionnaire collecte et filtre les données de gestion issues du niveau inférieur pour transmettre les informations pertinentes de façon synthétique au niveau supérieur.

Ce principe est utilisé dans le monde des télécommunications avec le modèle TMN (*Telecommunications Management Network*) [33] dans lequel chaque gestionnaire prend en charge un domaine d'administration. La hiérarchie des gestionnaires assure une certaine fiabilité lorsque la liaison est interrompue avec les gestionnaires de plus haut niveau en permettant aux gestionnaires de plus bas niveau, proches des agents, de prendre des actions correctives de façon autonome. De la même façon dans les réseaux IP, le besoin de structurer l'administration a conduit à une amélioration de SNMP permettant de déployer une hiérarchie de gestionnaires grâce aux *Manager-to-Manager MIBs* [26]. Cela a notamment permis aux grandes entreprises d'administrer l'intégralité de leur réseau informatique qui peut s'étendre sur de nombreux sites géographiquement distants.

2.2.3 Approches par politique

Les approches par politique sont une réponse à la complexification toujours croissante de l'activité d'administration. Le but est de fournir aux administrateurs des mécanismes de gestion automatiques leur permettant de s'abstraire des tâches de plus bas niveau pour se concentrer sur des objectifs de gestion supérieurs. Pour cela, les administrateurs définissent les actions à entreprendre en fonction d'une situation donnée grâce à des politiques. Les politiques sont un ensemble de règles qui s'expriment sous la forme « si condition alors action » [125].

L'*IETF Policy Framework working group* a défini une architecture qui sert de référence aux approches d'administration par politique [163]. La figure 2.7 présente cette architecture :

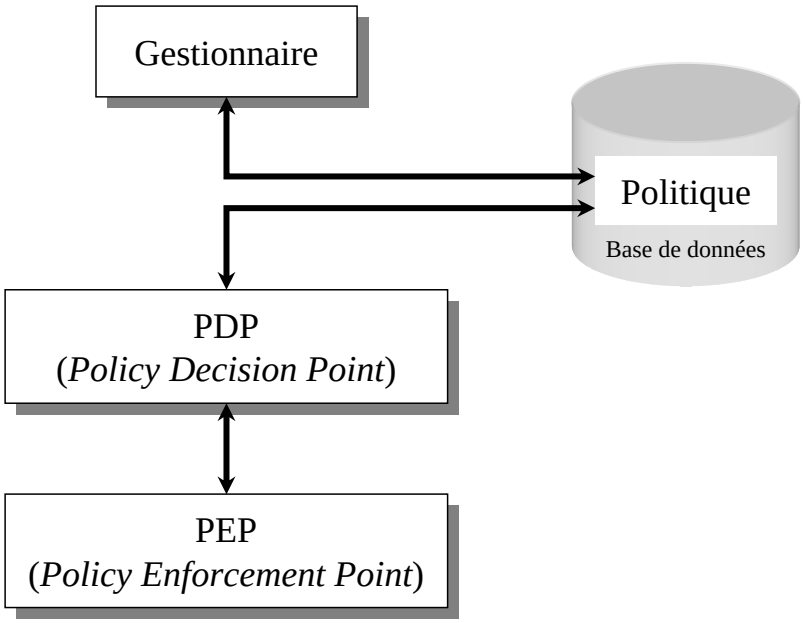


Fig. 2.7 – Approche d'administration par politique

Le PDP (*Policy Decision Point*) est l'entité qui prend les décisions politiques. C'est un organe central qui fait office de serveur de politiques, responsable de la gestion de requêtes d'administration et de la détermination des règles de politique à appliquer en fonction de ces requêtes. Pour prendre ses décisions, le PDP recherche les informations dont il a besoin dans une base de données qui répertorie les règles de politique. Cette communication utilise généralement le protocole LDAP (*Lightweight Directory Access Protocol*) [164, 158] qui est dédié à l'interrogation et à la modification de services d'annuaire. Une fois que le PDP a identifié quelle règle il faut appliquer, il se charge de la traduire en commandes interprétables par les ressources concernées et envoie ensuite ses ordres aux PEPs (*Policy Enforcement Points*).

Un PEP est une entité qui applique les décisions politiques prises par le PDP dont elle dépend. Il est responsable de l'exécution sur les ressources gérées des actions qui correspondent aux politiques qu'il reçoit. Un PEP peut également recevoir des notifications de la ressource nécessitant une décision politique, dans le cas d'une erreur par exemple. Il formule alors une requête pour une prise de décision politique qu'il envoie au PDP. En retour, le PDP lui renvoie les commandes appropriées à la gestion du problème.

2.2.4 Approches autogérées

Les approches autogérées vont encore plus loin que les approches par politique pour pallier la difficulté croissante de l'administration des systèmes informatiques. En effet, l'informatique autogérée[6] considère que la complexité de l'activité d'administration a atteint les limites des capacités humaines et qu'il faut donc non seulement que les systèmes d'administration fournissent des mécanismes d'automatisation du processus d'administration mais aussi que les systèmes eux-mêmes soient capables de s'administrer de façon autonomique, autrement dit de s'auto-administrer [55].

Dans leur vision [78], Kephart et Chess imagine l'informatique autogérée à partir d'une métaphore biologique. Ils s'inspirent du système nerveux végétatif qui contrôle sans aucun effort conscient les activités physiologiques du corps humain comme la respiration, la sudation, ou encore la pression sanguine. De là, ils définissent quatre propriétés générales des systèmes autogérés, connues sous le nom de propriétés *self-** (auto-*) :

- *self-configuration* (auto-configuration) : Les systèmes doivent être capables de s'adapter automatiquement aux modifications de leur environnement.
- *self-optimization* (auto-optimisation) : Les systèmes doivent constamment chercher à améliorer leurs performances.
- *self-healing* (auto-réparation) : Les systèmes doivent détecter, diagnostiquer et réparer les défaillances qui peuvent survenir.
- *self-protection* (auto-protection) : Les systèmes doivent anticiper, détecter, identifier et neutraliser les attaques qu'ils peuvent subir.

[6]Traduction de l'anglais *autonomic computing.*. On trouve également dans la littérature le terme francisé « informatique autonomique ».

Ainsi, un système autogéré se compose d'éléments d'autogestion responsables de l'administration de leur propre comportement et de leurs relations avec les autres éléments afin de maintenir les différentes propriétés *self-** de l'ensemble du système. La figure 2.8 représente la structure d'un élément d'autogestion interagissant avec les autres éléments du système autogéré.

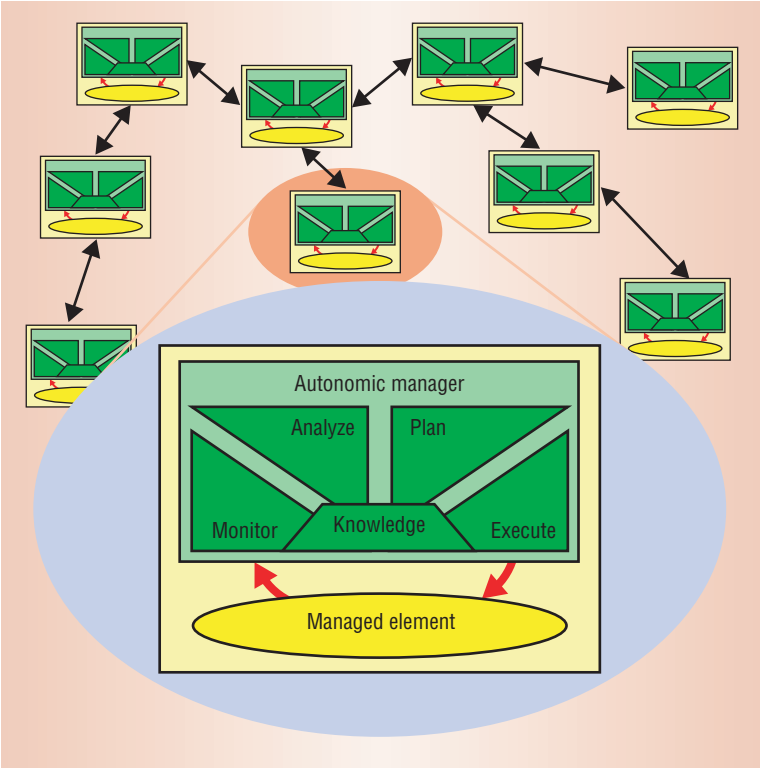


FIG. 2.8 – Approche d'administration autogérée (extrait de [78])

Un élément d'autogestion comprend un ou plusieurs éléments gérés et leur gestionnaire autonomique responsable de leur surveillance et de leur contrôle. On retrouve ici les principes fondamentaux de l'administration. Comme dans les approches non-autogérées, les éléments gérés doivent fournir les capteurs et les activateurs — renommés effecteurs pour se conformer à la métaphore biologique — nécessaires à leur instrumentation.

Ce qui distingue les approches autonomiques sont les mécanismes qui permettent aux gestionnaires de relier les activités de surveillance et de contrôle dans un processus autonome. Dans l'architecture de référence [67], un gestionnaire d'autogestion implémente une boucle de contrôle « intelligente » qui, à partir de la collecte des informations du système, est capable de décider et d'entreprendre les actions nécessaires au respect des propriétés *self-**. Cette boucle est composée des quatre fonctions suivantes :

- *monitor* (surveiller) : Il s'agit de collecter, d'associer et de filtrer les informations des ressources administrées afin de constituer des symptômes analysables.

- *analyze* (analyser) : Les symptômes transmis par la fonction précédente sont confrontés à des modèles complexes permettant de déterminer si des changements sont nécessaires.
- *plan* (planifier) : Lorsque des changements doivent être opérés, cette fonction construit et sélectionne les opérations nécessaires pour appliquer les modifications désirées.
- *execute* (exécuter) : Cette fonction exécute sur les ressources les actions dictées par le plan.

La conception d'un système réellement autonomique est encore immature et risque de se confronter à des problèmes relevant de l'intelligence artificielle. Néanmoins cette vision a le mérite d'amener les recherches sur l'administration des systèmes vers plus d'automatisation et d'élever le niveau d'abstraction des informations de gestion comme nous le verrons avec l'administration des composants logiciels dans la section suivante.

2.3 Administration des composants logiciels

Les caractéristiques des composants logiciels que nous avons présentées au chapitre précédent influent particulièrement sur l'administration des systèmes à composants [156]. L'encapsulation, par exemple, qui cache les détails internes des composants, rend leur instrumentation complexe. Le système d'administration doit également prendre en compte les principes de composition du modèle de composants sous-jacent, de même que l'infrastructure dans laquelle sont déployés les composants.

Malgré cette complexité supplémentaire, le paradigme des composants logiciels peut en retour s'avérer bénéfique pour appréhender le processus global de l'administration puisqu'il décompose le système en unités individuellement administrables comme le suggère Szyperski :

> *« Certain parts of a system may require explicit management that goes beyond maintenance [...]. To manage a system, it is useful to partition it into units of management, each of which can be individually monitored for availability, load, and so on. »* [154]

Dans cette section, nous nous intéressons aux différentes approches d'administration des composants logiciels, à savoir :

- les approches qui administrent les composants au niveau de l'infrastructure dans laquelle ils sont déployés,
- les approches qui permettent de reconfigurer l'architecture des systèmes à base de composants logiciels,
- les approches qui prennent en compte le comportement même des composants.

2.3.1 Approches basées sur l'infrastructure

JMX (*Java Management eXtensions*) [145] est devenu un standard incontournable pour l'administration des composants logiciels. Dans cette section, nous présentons tout d'abord JMX,

ensuite nous nous intéressons à l'administration des composants par JMX dans les serveurs d'applications Java EE puis dans les infrastructures d'autres modèles de composants.

Java Management eXtensions

La spécification de *Java Management eXtensions* (JMX) [145] définie une architecture d'administration et un ensemble complet d'interfaces de programmation permettant d'administrer des équipements, des réseaux, des applications, ou des services via la technologie Java [86]. Issue du *Java Community Process* (JCP), JMX est l'initiative d'un consortium industriel regroupant entre autres Sun Microsystems, IBM, Computer Associates, BEA Systems, Apache Software Foundation, Bull S.A., Alcatel-Lucent, et Motorola. Depuis sa première distribution en tant que paquetage optionnel en 1999, JMX a connu un franc succès et fait maintenant partie intégrante de Java SE depuis la version 5.0.

L'architecture de JMX s'appuie sur le modèle classique de « gestionnaire - agent » défini dans le cadre général de l'administration (cf. 2.4 p. 40). On retrouve ainsi dans JMX les trois entités fondamentales de ce modèle, c'est-à-dire le gestionnaire, l'agent et les objets gérés. La figure 2.9 présente l'architecture de JMX et situe la place de ses composants dans les différents niveaux qu'elle comprend.

Cette architecture distingue trois niveaux :

- *Instrumentation Level* (niveau de l'instrumentation) : ce niveau comprend les ressources administrées et s'intéresse en particulier aux moyens de les instrumenter.
- *Agent Level* (niveau des agents) : c'est le niveau qui concernent les agents et notamment les services qu'ils fournissent pour administrer les ressources.
- *Distributed Services Level* (niveau des services distribués) : ce niveau devrait être le niveau des gestionnaires mais la spécification de JMX n'a pas pour but de définir comment construire les gestionnaires, seulement comment les agents communiquent avec eux.

Le concept de base de l'architecture de JMX est le *Managed Bean* ou MBean, autrement dit un composant Java administré. Le MBean représente les ressources que le système d'administration doit observer et contrôler. Concrètement, un MBean est implémenté par un objet Java qui implémente une interface d'administration exposant les services administrables de la ressources et ses attributs, présentés selon les méthodes *getter* & *setter* des JavaBeans (cf. 1.1 p. 13). Si la communication entre le MBean et la ressource qu'il représente induit pour des éléments non-Java l'utilisation de mécanismes complexes tels que *Java Native Interface* (JNI) ou d'autres adaptateurs, l'instrumentation des applications Java, et en particulier les composants qui suivent les conventions de nommage des JavaBeans, est directe puisqu'il suffit de décrire une simple interface Java.

JMX spécifie cinq types de MBeans :

- les *Standard MBeans* : l'interface d'administration des *Standard MBeans* est définie une fois pour toute lors du développement. Elle correspond à une simple interface Java comme

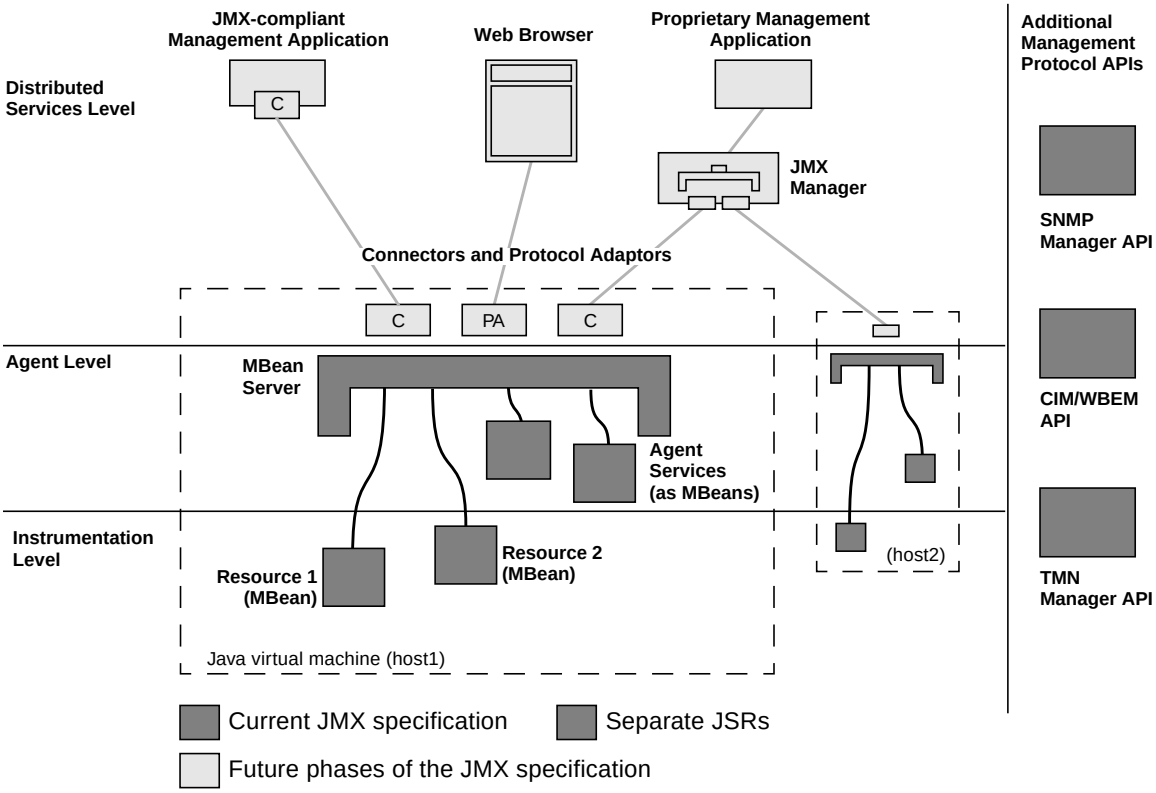


FIG. 2.9 – Architecture de JMX (extrait de [145])

le montre l'exemple du listing 2.1. Les différents éléments de cette interface, *i.e.* méthodes et attributs, sont retrouvés par réflexion par le système d'administration de JMX.

- les *Dynamic MBeans* : l'interface d'administration des *Dynamic MBeans* est construite de façon programmatique. Elle est donc générée à l'exécution et peut également être modifiée et adaptée en fonction de l'évolution de la ressource administrée.
- les *Open MBeans* : les *Open MBeans* sont des *Dynamic MBeans* qui sont limités dans leur utilisation des types de données pour permettre une interopérabilité maximale avec les systèmes d'administration non-Java.
- les *Model MBeans* : les *Model MBeans* sont des *Dynamic MBeans* génériques et configurables. Ils sont une solution alternative aux *Dynamic MBeans* souvent fastidieux à implémenter.
- les *MXBeans* : les *MXBeans* correspondent à des *Open MBeans* mais créés sur le modèle des *Standard MBeans*. Introduits avec Java SE 5.0, ils sont utilisés par les *platform MBeans* pour administrer les composants de la machine virtuelle Java. Ainsi la plate-forme rend administrable, entre autres composants, son système de chargement de classes par l'interface `ClassLoadingMXBean`, son système de mémoire par l'interface `MemoryMXBean`, le ramasses-miette par l'interface `GarbageCollectorMXBean` et son système de *threads* par l'interface `ThreadMXBean`.

```
1  // Interface du standard MBean Server
2  public interface ServerMBean {
3      void shutdown();
4      void start();
5      void stop();
6      int  getConnections();
7      int  getMaxConnections();
8      void setMaxConnections(int max);
9      long getUptime();
10 }
```

List. 2.1 – Interface d'administration du MBean `Server`

Tout comme les objets gérés sont concentrés au niveau d'un agent dans le modèle « gestionnaire - agent », les MBeans sont intégrés dans un agent JMX avec lequel ils peuvent communiquer via un système de notifications. Pour être administrés par un agent, les MBeans doivent être déployés sur la même JVM que l'agent. L'élément central de l'agent est le serveur de MBean, nommé *MBean Server*, dans lequel sont enregistrés les MBeans par un nom unique de la forme :

```
1      [DomainName]:property=value[, property = value]*
```

Depuis Java 5, la JVM intègre un *MBean Server* particulier, le *Platform MBean Server*. Il permet d'administrer les différents composants de la JVM mais il peut être également utilisé pour administrer tous les MBeans s'exécutant sur la JVM. On y accède par la méthode `ManagementFactory.getPlatformMBeanServer()` du paquetage `java.lang.management`.

L'agent comprend aussi un ensemble de services eux-mêmes déployés sous forme de MBeans :

- *Monitor Service* : les moniteurs permettent d'observer facilement des attributs en déclenchant par exemple des alarmes lorsqu'un attribut numérique dépasse un certain seuil.
- *Timer Service* : ce service permet de réaliser automatiquement des opérations périodiques par l'utilisation de *Timers*.
- *Dynamic MBean Loading Service* : ce service permet de charger à distance de nouveaux MBeans.
- *Relation Service* : le service de relation permet de gérer les dépendances entre les MBeans pour lier par exemple leur cycle de vie.
- *JMX Connectors & Protocol Adaptors* : JMX fournit un ensemble de connecteurs et d'adaptateurs pour relier l'agent aux gestionnaires distants, notamment via HTTP, RMI, SNMP, CIM/WBEM ou encore TMN.

Comme dans les approches « gestionnaire - agent » classiques, l'agent JMX est responsable de la liaison entre le gestionnaire et les ressources administrées, ici les MBeans. Cependant, l'agent JMX n'est pas limité à un simple relais, il peut profiter de toute la puissance de Java pour réaliser des opérations d'administration complexes. Ainsi, les gestionnaires se limitent souvent à un rôle de présentation tandis que l'agent assure tout le processus d'administration. Un simple navigateur Web suffit alors pour se connecter à un agent JMX afin d'opérer les tâches d'administration à distance. La plate-forme Java fournit dorénavant une console graphique un peu plus évoluée qu'un navigateur Web, la *JConsole*, qui permet d'administrer des serveurs JMX via RMI. JMX peut néanmoins s'intégrer parfaitement dans des applications d'administration complexes telles que IBM Tivoli ou HP OpenView.

JMX et Java EE

L'administration est une activité cruciale pour la plate-forme Java EE qui met en œuvre de nombreux composants et mécanismes complexes. Dans sa spécification, le rôle de l'administrateur système est défini comme suit :

> *« The System Administrator is responsible for the configuration and administration of the enterprise's computing and networking infrastructure that includes the EJB server and container. The System Administrator is also responsible for overseeing the well-being of the deployed enterprise beans applications at runtime. »* [147, 150]

L'administrateur doit tout d'abord pouvoir gérer le serveur d'applications, c'est-à-dire l'infrastructure d'accueil des composants elle-même. Pour cela, les premiers vendeurs de serveurs d'applications proposaient des solutions *ad hoc* minimales en fonction de leur implémentation, rendant ainsi impossible l'administration unifiée de serveurs différents. Dès sa sortie, JMX a rapidement été adopté par la communauté Java EE mais malgré une uniformisation *de facto* sur la technologie, il existait toujours des incompatibilités dans les mécanismes proposés. Aujourd'hui,

le JSR 77 [65] définit un cadre pour l'administration de Java EE et JMX est le support de choix pour l'implémenter [50]. D'ailleurs, pratiquement tous les serveurs actuels comme BEA WebLogic, Borland AppServer, JBoss, ou Glassfish, proposent une solution d'administration basée sur JMX.

Le JSR 77 décrit donc un modèle d'information pour l'administration des serveurs Java EE, les composants applicatifs EJB et les ressources qui composent les services standards de l'infrastructure de Java EE. La figure 2.10 présente l'ensemble des composants administrables et leur modèle de base commun, le `J2EEManagedObject`.

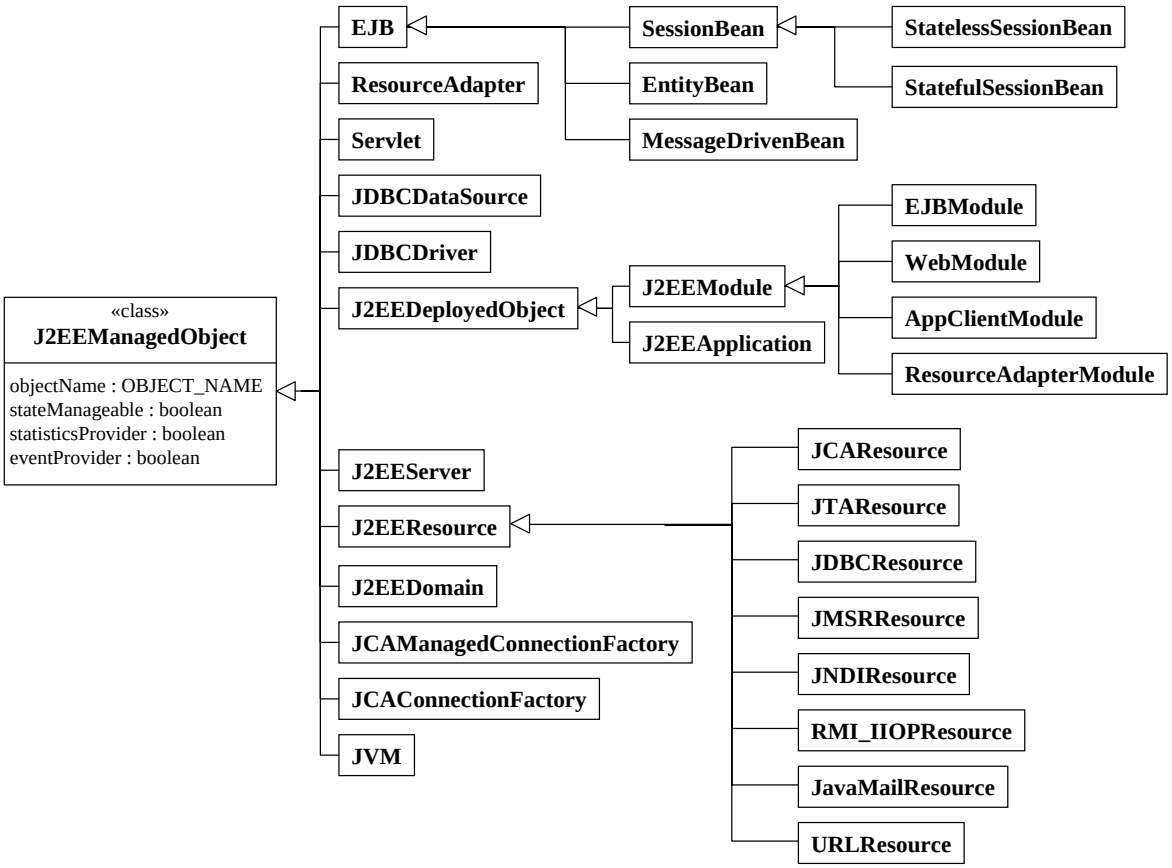


FIG. 2.10 – Unités d'administration des serveurs Java EE (JSR 77)

Un composant conforme au modèle `J2EEManagedObject` doit donc posséder un identificateur unique et indiquer s'il peut fournir des informations d'administration relatives à son état de cycle de vie, à ses performances ou aux événements qu'il peut transmettre. Pour l'instrumentation de ces trois critères, le JSR 77 propose des interfaces d'administration standardisées. Par exemple, l'interface `StateManageable` présentée en figure 2.11 permet d'administrer les composants de la plate-forme selon la machine à état de la figure 2.12. Ce comportement, certes général, peut suffire à certains composants simples de l'infrastructure mais n'est pas adéquat pour des composants applicatifs complexes. D'ailleurs, cette machine à état ne couvre pas totalement le cycle de vie

qu'impose l'infrastructure Java EE aux EJB.

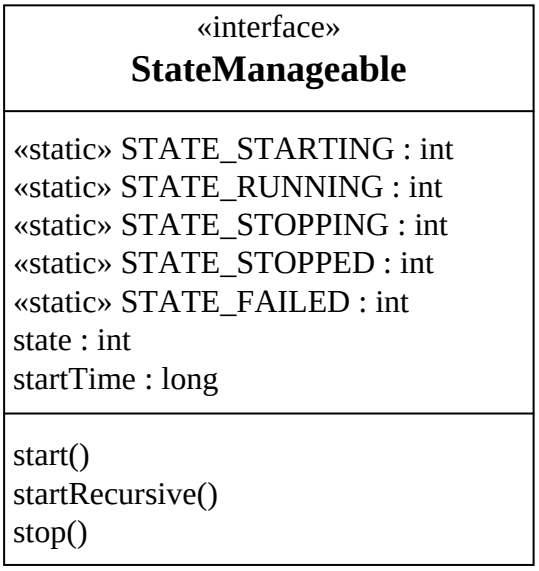


FIG. 2.11 – Interface `StateManageable` du JSR 77

L'instrumentation des EJB nécessite des interfaces spécifiques, prenant en compte les particularités métier des composants. Plusieurs travaux ont proposé des moyens pour instrumenter les EJB, manuellement [6] ou même automatiquement [3]. Dans ces approches, un EJB est représenté dans l'agent JMX par un MBean. Le MBean, qui est en relation avec l'EJB, y expose alors un sous-ensemble des services fournis par les interfaces Home et Remote du composant en plus d'autres méthodes d'administration. Le diagramme de la figure 2.13 en décrit l'architecture classique correspondante.

JMX, un standard pour l'administration des composants logiciels

A l'instar des EJB, de nombreux modèles de composants incorporent dans leurs composants une interface d'administration suivant le modèle des MBeans afin de permettre une administration via JMX. Parmi ces modèles, on peut citer notamment Spring, NanoContainer, HiveMind, Carbon, Loom, OSGi ou encore Fractal.

L'intégration de l'agent JMX dans les infrastructures à composants correspondantes peut suivre trois stratégies différentes identifiées par Kreger [86] :

- *daemon* : l'agent et l'application à composants s'exécutent dans des processus différents. C'est l'approche traditionnelle des systèmes d'administration « gestionnaire - agent ».
- *component* : l'agent est intégré comme un composant du système.
- *driver* : l'agent sert d'infrastructure à l'application qui s'exécute au sein de l'agent.

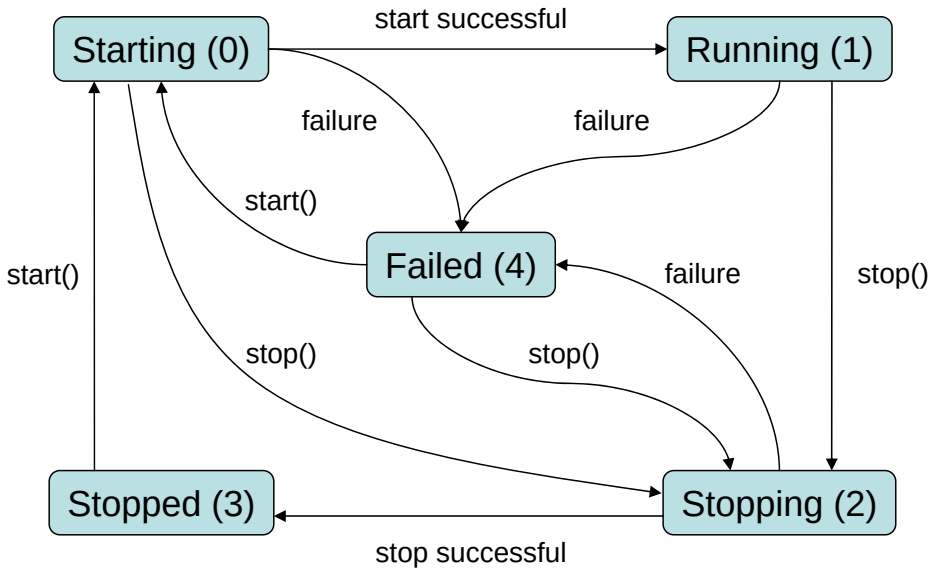


FIG. 2.12 – Le modèle des états administrables du JSR 77

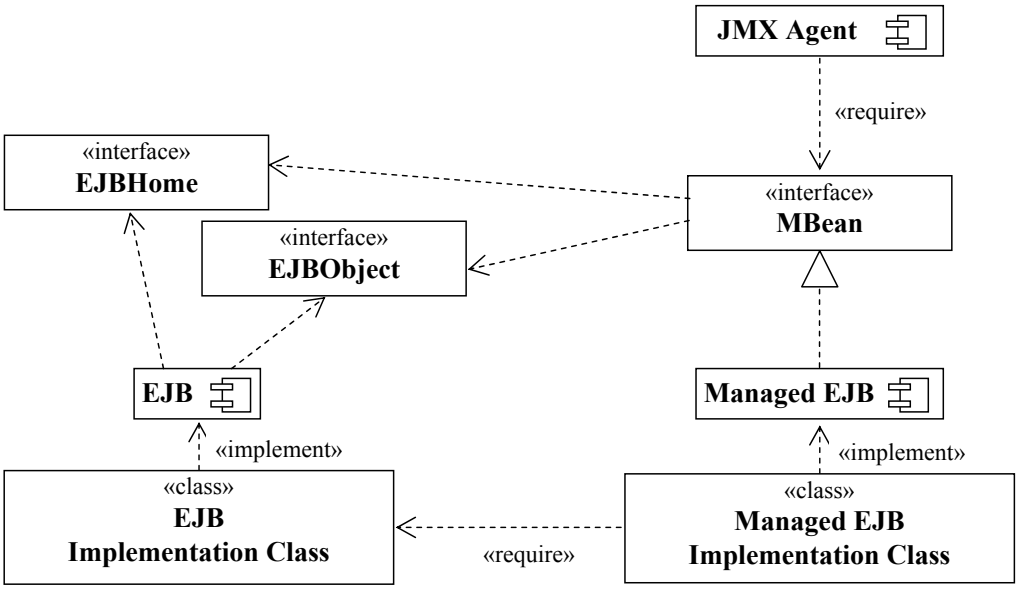


FIG. 2.13 – Architecture classique de l'administration des EJB 2.x avec JMX

L'approche *daemon* est la plus couramment utilisée surtout depuis que la machine virtuelle Java propose par défaut son *Platform MBeanServer* pour l'administration JMX. Au contraire, l'approche *driver* n'est que rarement utilisée car elle implique que le support de JMX soit pris en compte dès le début de la conception. Cela peut cependant être une solution intéressante pour les modèles légers qui ne possèdent pas d'infrastructure.

Certains modèles comme Fractal et OSGi mettent à profit l'approche *component* pour intégrer JMX dans leur infrastructure. Ainsi dans [52], JMX est intégré directement comme un service délivré comme un bundle OSGi. De son côté, Fractal JMX [127] propose un agent JMX sous forme de composant Fractal qui s'intégre au sein des application elles-mêmes et utilise les mécanismes d'introspection du modèle pour instrumenter directement toute la hiérarchie des composants de l'application.

2.3.2 Approches basées sur l'architecture

L'aspect boîte noire des composants logiciels et la définition explicite de leurs services fournis et requis facilitent leur substituabilité. Un composant peut donc être remplacé par un autre composant respectant les mêmes contraintes externes. Bien qu'interchangeables dans certaines conditions, les composants possèdent leur propre logique interne, et le remplacement d'un composant par un autre ou des modifications dans leurs connexions induisent forcément des changements sur l'application globale. L'administration de l'architecture d'une application à composants est donc un moyen d'agir indirectement sur son comportement.

Il existe de nombreuses propositions qui permettent de reconfigurer ou d'adapter une application en s'appuyant sur son architecture à composant. La reconfiguration et l'adaptation sont d'ailleurs des problématiques à part entière du génie logiciel basé composant. Nous avons choisi de présenter dans cette section deux solutions issues de travaux académiques récents qui capturent l'essence même de ces deux approches et sont en même temps des sources référençant un état de l'art plus spécifique.

DYVA

DYVA, qui est l'acronyme de *DYnamic Virtual Adaptation machine*, est un système de reconfiguration dynamique pour les applications à base de composants, défini dans le cadre de la thèse de Ketfi [79] à l'Université Joseph Fourier de Grenoble. Ce système utilise les principes de la réflexivité [140] pour séparer l'application de la représentation abstraite de son architecture. Ainsi comme le montre la figure 2.14, l'application dans son environnement d'exécution constitue le niveau de base tandis que le système de reconfiguration opère au niveau méta sur l'architecture abstraite de l'application. Les niveaux de base et méta sont causalement liés, c'est-à-dire que les modifications du système de reconfiguration sur l'image abstraite de l'application sont projetées sur l'application elle-même et inversement.

Le niveau méta de DYVA définit un ensemble de concepts architecturaux que l'on retrouve typiquement dans les ADL, et dont voici la liste :

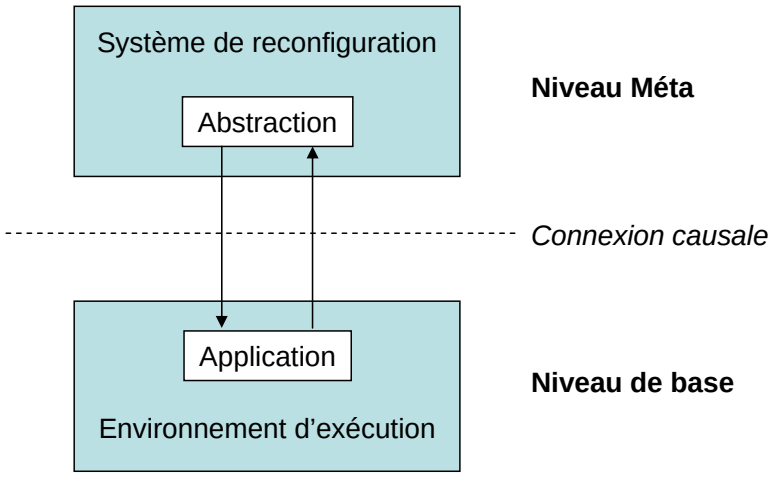


FIG. 2.14 – Conception réflexive de DYVA

- composant,
- composant primitif,
- composant composite,
- port fourni,
- port requis,
- instance de composant.

Ces éléments permettent aux concepteurs de DYVA de modéliser, ou plutôt de métamodéliser, l'architecture des applications indépendamment du modèle de composants employé. Ainsi les fonctionnalités de reconfiguration de DYVA s'appliquent seulement sur un modèle abstrait et l'ajout d'un *plugin* dans le système permet d'établir la correspondance entre ce modèle abstrait et le modèle de composants considéré. Cela permet d'administrer les applications à composants de manière uniforme et permet également de créer des fonctions de projection pour les modèles qui ne sont pas réflexifs.

L'architecture du système de gestion de DYVA est représentée en figure 2.15. Elle comprend donc le modèle abstrait ainsi qu'un superviseur responsable de l'observation du modèle et un gestionnaire de reconfiguration dont la responsabilité est d'effectuer les modifications nécessaires sur le modèle abstrait. Ces modifications sont décidées par le superviseur d'après les politiques de reconfiguration définies dans le système. Ces politiques comprennent un ensemble de règles qui sont décrites suivant un formalisme ECA (*Event-Condition-Action*) utilisé dans les bases de données actives [35, 43]. Le listing 2.2 montre un exemple de règle dans le système DYVA.

```
1 [RULE]
2 ON: OVERLOAD_EVENT
3 IF: ((LOAD > 80) AND (Handler.version = 1))
4 THEN: replace Handler1 NewHandler1
```

LIST. 2.2 – Exemple de règle de reconfiguration dans DYVA

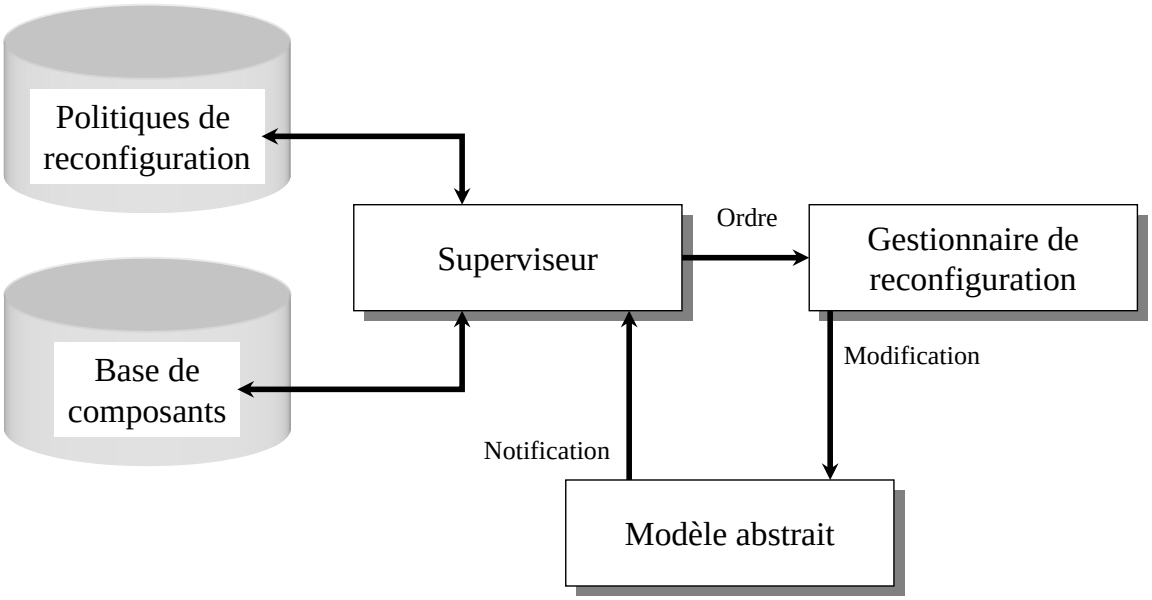


Fig. 2.15 – Architecture de DYVA

A la réception d'une notification, le superviseur vérifie s'il existe un règle correspondante à l'événement reçu (clause `ON`). Si une telle règle existe, le superviseur vérifie si la clause `IF` de la règle est vérifiée. Si tel est le cas, il ordonne au gestionnaire de reconfiguration d'exécuter les actions de clause `THEN` de la règle.

Les possibilité de reconfiguration offertes par le gestionnaire de reconfiguration sont :

- la passivation d'une instance de composant,
- l'activation d'une instance de composant,
- la suppression d'une instance de composant,
- la suppression d'un composant,
- la déconnexion dynamique,
- la connexion dynamique,
- la reconnexion dynamique,
- l'ajout d'un composant,
- la création d'une instance de composant,
- le remplacement d'une instance de composant.

Pour le remplacement de composants, le système DYVA propose un mécanisme de transfert de l'état des composants. L'état est défini comme un ensemble d'attributs accessibles par des méthodes *getter* et *setter*. Lors d'un remplacement le système sauvegarde la valeur de ces attributs et l'affecte au nouveau composant de sorte que l'application puisse continuer son exécution en cours.

Ce mode de reconfiguration implique cependant une hypothèse forte : il doit exister un composant permettant de répondre aux nouveaux besoins. Pour cela le système dispose d'une

base de composants dans laquelle il peut rechercher de nouveaux composants à insérer dans l'architecture en place, mais rien ne dit que cette recherche sera fructueuse.

SAFRAN

SAFRAN propose une extension du modèle de composants Fractal permettant d'associer dynamiquement des politiques d'adaptation aux composants d'une application. SAFRAN signifie *Self-Adaptive FRActal compoNents* et cette extension de Fractal est issue des travaux de thèse de David à l'Ecole des Mines de Nantes [40].

Par rapport au système de reconfiguration que nous avons étudié précédemment, SAFRAN s'intéresse plus à l'adaptation des composants, c'est-à-dire qu'ils doivent non seulement être reconfigurables mais ils doivent également être sensibles à leur contexte d'exécution. Pour cela, SAFRAN intègre le système WildCAT qui permet de représenter le contexte de l'application afin de l'observer à l'exécution. La figure 2.16 décrit le modèle de représentation du contexte dans WildCAT.

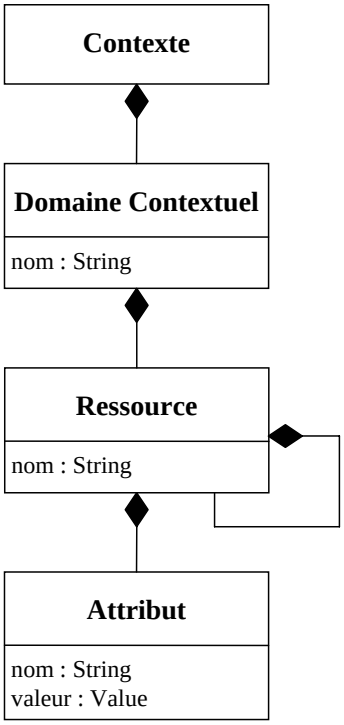


Fig. 2.16 – Modélisation des ressources dans SAFRAN

Dans SAFRAN, la surveillance assurée par WildCAT est opérée sur des ressources externes de l'application telles que la mémoire ou le système de fichiers du système ou des ressources plus abstraites comme la localisation de l'utilisateur par exemple. Une ressource est caractérisée

par un certain nombre d'attributs, c'est-à-dire des couples (*nom*, *valeur*) donnant des informations spécifiques sur la ressource. L'évolution du contexte correspond alors à des modifications des valeurs des attributs, à l'apparition ou la disparition de ressources ou d'attributs dans les ressources.

La surveillance des composants de l'application ne suit pas le modèle de représentation que les ressources mais utilise les capacités d'introspection du modèle Fractal. Ainsi, des informations peuvent être obtenues sur l'échange de messages entre composants par leurs interfaces de service, sur le cycle de vie des composants par le contrôleur Fractal `LifeCycleController`, sur la modification des attributs via le contrôleur `AttributeController` ou sur les sous-composants d'un composite via son contrôleur `ContentController`.

SAFRAN définit un nouveau contrôleur Fractal nommé `AdaptationController` pour associer individuellement aux composants des politiques d'adaptation. Ces politiques ne peuvent agir qu'au niveau du composant auquel elles sont associées, c'est-à-dire sur le composant lui-même et sur ses sous-composants, Fractal étant un modèle hiérarchique. Les politiques d'adaptation sont décrites à l'aide d'un langage de script, appelé FScript. De nouvelles politiques peuvent ainsi être appliquées dynamiquement à l'exécution via une console textuelle.

Comme dans DYVA, l'approche que nous avons étudiée précédemment, les règles d'adaptation suivent un formalisme ECA [35, 43]. Le listing 2.3 montre un exemple d'une telle règle dans le langage FScript.

```
1 rule enforce_minimum = {
2   when e:subcomponent-removed($target)
3   if (count($target/child::*) < 10)
4   do {
5     start($target/child::*);
6   }
7 }
```

List. 2.3 – Exemple de règle d'adaptation dans SAFRAN

Pour le contrôle des composants, SAFRAN s'appuie sur les capacités d'intercession du modèle de composants Fractal. Les règles FScript utilisent donc les contrôleurs Fractal notamment le contrôleur de liaisons `BindingController` et le contrôleur de contenu `ContentController` pour opérer des reconfigurations sur l'architecture de l'application. SAFRAN, comme DYVA, présuppose donc la même hypothèse forte sur l'existence d'un composant ou d'un nouvel assemblage de composants permettant de répondre aux nouveaux besoins.

2.3.3 Approches basées sur le comportement

Face à la part d'imprévisibilité qui réside dans un assemblage logiciel, l'un des enjeux majeurs de l'administration des composants est de s'assurer que le comportement des composants

respectent leur spécification à l'exécution [152]. Dans la littérature, les approches qui s'intéressent à l'administration du comportement interne des composants sont peu nombreuses et se concentrent en particulier sur l'aspect surveillance. La figure 2.17 décrit l'architecture générale qu'elles emploient.

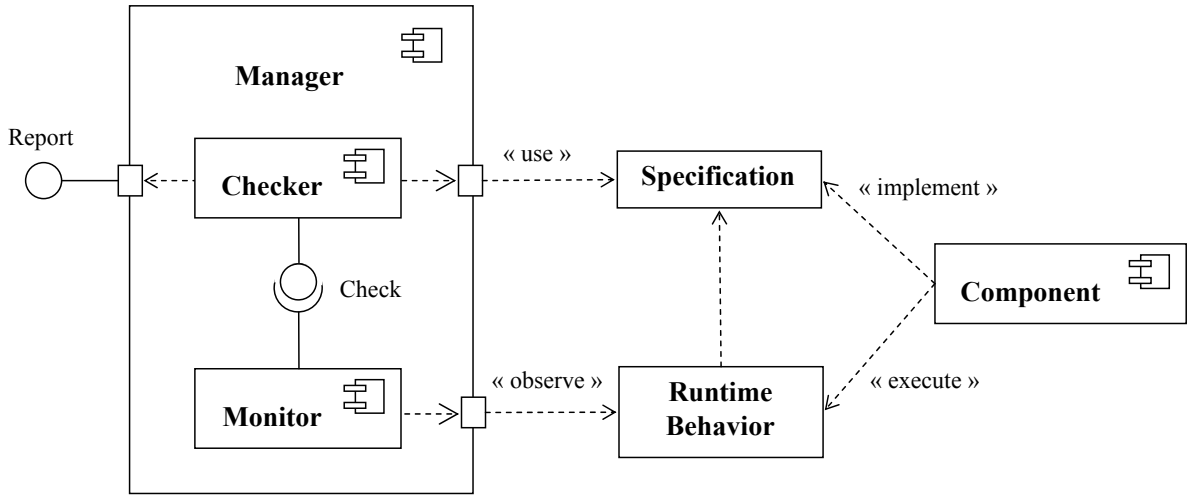


FIG. 2.17 – Architecture générale des approches basées sur le comportement

Le système d'administration de cette architecture comprend généralement un *Monitor* pour observer à l'exécution le comportement du composant cible et un *Checker* avec lequel le *Monitor* communique pour vérifier que l'évolution du comportement du composant satisfait toujours sa spécification. Le *Manager* doit donc pouvoir avoir accès au comportement du composant et à sa spécification.

Ces approches se différencient principalement par :

- le formalisme de spécification employé,
- la méthode de réification du comportement à l'exécution,
- les techniques d'instrumentation permettant d'observer ce comportement,
- la nature des problèmes détectables.

Dans cette section, les approches que nous présentons sont étudiées selon ces quatre critères que nous détaillons pour chacune dans les paragraphes intitulés *Spécification*, *Comportement*, *Instrumentation* et *Détection d'erreur*.

Surveillance du comportement des composants avec TLA

Spécification. Les auteurs de [37, 128] définissent une approche de surveillance du comportement des composants basée sur une spécification du système en TLA (*Temporal Logic of Action*) [87]. TLA est utilisé pour décrire des systèmes concurrents et réactifs. Ce formalisme combine

une logique d'action qui est définie en termes d'états et une logique temporelle qui permet de résonner sur le comportement du système en termes de séquences d'états. Les états correspondent à l'affectation de valeurs à des variables. TLA est particulièrement efficace pour la vérification de modèles (*model checking*).

Comportement. Pour permettre sa vérification à l'exécution, la spécification du système est logiquement composée avec des observateurs, c'est-à-dire des formules TLA utilisant des opérateurs permettant d'acquérir des valeurs du système à l'exécution. Les observateurs reflètent donc le comportement réel du système et la spécification son comportement attendu. Cette approche permet ainsi d'utiliser un *model checker* classique pour vérifier le système à l'exécution. Le *model checker* n'évalue plus le comportement du système seulement à partir de valeurs provenant d'un modèle mais également à partir de valeurs du système en cours d'exécution.

Instrumentation. Dans [37], les auteurs évoquent la possibilité de définir un contrôleur Fractal pour supporter leur approche, mais ils ne définissent pas comment lier leur *monitor* avec les composants du système.

Détection d'erreur. Les techniques de *model checking* permettent de vérifier des propriétés temporelles des systèmes telles que la sûreté, la vivacité ou l'équité, mais cette approche de vérification à l'exécution ne peut traiter que les propriétés de sûreté.

Surveillance du comportement des composants avec les CFSM

Spécification. Dans [167], le comportement des composants, ou d'un assemblage de composants, est spécifié avec des automates à état communicants, des CFSM (*Communicating Finite-State Machines*) [21]. Par rapport aux automates à état traditionnels, ces automates disposent en plus de canaux de communication entrants et sortants leur permettant de communiquer entre eux et avec l'environnement système par envoi de messages.

Comportement. Le comportement des composants est perçu à travers les entrées et les sorties des composants et leurs états stables. Les entrées et les sorties d'un composant correspondent à des événements spécifiés dans son CFSM entre le composant et son environnement. Un composant est dit dans un état stable lorsqu'il est en attente d'un événement extérieur et que son exécution ne peut pas évoluer sans une telle entrée. Lors du passage d'un état stable à un autre, le comportement du composant peut emprunter différents chemins dans son CFSM, qui correspondent à des transitions impliquant des événements internes et des états non stables. Le comportement entre deux états stables n'étant pas observé, le système d'administration utilise une stratégie *Assume/Guarantee* [75] pour déterminer les comportements possibles, c'est-à-dire générer les différentes trajectoires possibles en terme de séquences d'événements dans le CFSM, et les confronter aux observations effectuées sur le système global.

Instrumentation. Dans cette approche, l'instrumentation est peu intrusive puisqu'elle s'appuie principalement sur l'observation des événements externes des composants. Le *monitor* est

alors responsable de capturer les entrées et les sorties aux interfaces des composants. Néanmoins, les composants doivent être capables de notifier leurs changements d'états (stables) mais les auteurs de [167] ne précisent pas comment cela est opéré.

Détection d'erreur. La détection d'erreur est effectuée à plusieurs niveaux :

- au niveau du composant lui-même, lorsqu'il se trouve dans un état qui ne peut pas être atteint à partir de son état précédemment observé,
- au niveau de la génération des trajectoires possibles, lorsque le nouvel état ne peut pas être atteint avec les événements précédemment observés,
- au niveau de la communication entre les composants, lorsqu'il n'est pas possible de trouver pour chaque composant une trajectoire possible satisfaisant l'ensemble des comportements des composants.

Surveillance du comportement avec les réseaux de Petri

Spécification. L'approche utilisée dans [60] emploie les réseaux de Petri [123] pour spécifier le comportement des composants. Les places du réseau de Petri représentent l'état du composant. Les transitions, étiquetées avec les messages échangés, modélisent les interactions avec les autres composants. Cette approche définit de plus trois types de places :

- *waitfor*, pour spécifier que le composant est dans un état d'attente d'un message d'un autre composant,
- *calculation*, pour décrire un état dans lequel le composant est actif,
- *donothing*, pour introduire des états d'inactivité nécessaire à des fins de synchronisation,

et deux types d'arcs :

- *data_flow*, pour permettre de lier les messages d'une même transaction,
- *multi_exec*, pour décrire la synchronisation entre différentes transactions.

Comportement. La surveillance des composants n'est pas effectuée directement à partir des réseaux de Petri. Selon une transformation adaptée de l'approche définie dans [2], le comportement des composants est traduit en *tiles* qui sont des morceaux du réseau de Petri correspondant aux différentes transitions. Chaque *tile* est composé d'une précondition, d'une étiquette correspondant à l'événement associé, et d'une postcondition. Les pré et post conditions contiennent des variables qui remplacent les places du réseau de Petri pour représenter l'état du composant avant et après la transition. Le but de cette transformation du réseau de Petri est d'obtenir une représentation uniforme du comportement à l'exécution afin de gérer pour chaque composant un historique des transitions tirées.

Instrumentation. Dans cette approche, l'instrumentation s'effectue seulement sur les événements externes des composants, c'est-à-dire les échanges de messages entre les différents composants. Cela nécessite seulement d'installer des intercepteurs sur les interfaces des composants pour notifier au système d'administration de l'occurrence d'un message. Chaque message est estampillé par un marqueur temporel et il est ajouté à l'historique du composant s'il correspond à un *tile*.

Détection d'erreur. Comme cette approche ne se base que sur les messages externes échangés entre les composants, il est difficile lors de l'apparition d'un message d'erreur de connaître le composant réellement fautif. La cause du message d'erreur peut être imputée au composant qui l'émet ou elle peut être la conséquence d'une requête erronée provenant d'un autre composant. Pour cela, l'historique permet d'établir des taux d'erreur pour chaque composant et de suivre leur évolution afin de déterminer les composants véritablement défaillants.

2.4 Solutions d'administration des composants logiciels sans fil

L'administration du logiciel et en particulier des composants logiciels déployés dans les appareils sans fil revêt une importance croissante dans l'industrie. De nombreuses multinationales et autres entreprises impliquées dans le domaine du sans fil s'organisent en consortium pour établir des standards et proposer de nouvelles solutions pour l'administration sans fil.

Cet engouement industriel est cependant peu relayé au niveau académique. La littérature ne référence que peu de travaux de recherche spécifiquement sur cette problématique et les principaux efforts se concentrent plus sur des problématiques liées à la gestion de la mobilité (mobilité du code entre les appareils ou mobilité des appareils eux-mêmes) que nous n'abordons pas dans cette thèse.

Dans cette section, nous présentons les avancées de deux standards industriels concernant l'administration sans fil puis nous nous intéresserons à un travail de recherche original permettant la construction d'un *middleware* administrable à base de composants dans les systèmes sans fil.

2.4.1 OMA Device Management

L'*Open Mobile Alliance* (OMA) est un organisme de normalisation qui a été créé en juin 2002 pour développer des standards facilitant l'interopérabilité des appareils sans fil dans l'industrie du mobile. L'un des groupes de travail les plus actifs de l'OMA est le *Device Management Working Group*. Ce groupe travaille sur un ensemble de spécifications concernant l'administration des appareils sans fil [114].

Le modèle fonctionnel de ces spécifications regroupe notamment les activités suivantes :
- l'installation d'informations de configuration dans les appareils sans fil,
- la mise à jour des informations persistantes dans les appareils sans fil,
- la récupération d'informations d'administration au sein des appareils sans fil,
- le traitement des événements et des alarmes générés par les appareils sans fil.

Les informations d'administration concernent aussi bien les paramètres de configuration du fonctionnement de l'appareil lui-même, que le logiciel et les applications installés, ou encore les données personnelles de l'utilisateur. Le but principal est d'affranchir l'utilisateur final des procédures complexes de configuration des appareils sans fil en permettant à des tiers, comme par exemple les opérateurs sans fil, les fournisseurs de services ou les départements informatique des entreprises, de le faire.

Architecture

La technologie définie par OMA DM permet à un appareil de présenter ses informations d'administration à un serveur externe via une communication sans fil. La figure 2.18 présente cette architecture. Sur la gauche, on trouve l'appareil avec ses ressources internes et sur la droite le serveur d'administration avec lequel il communique.

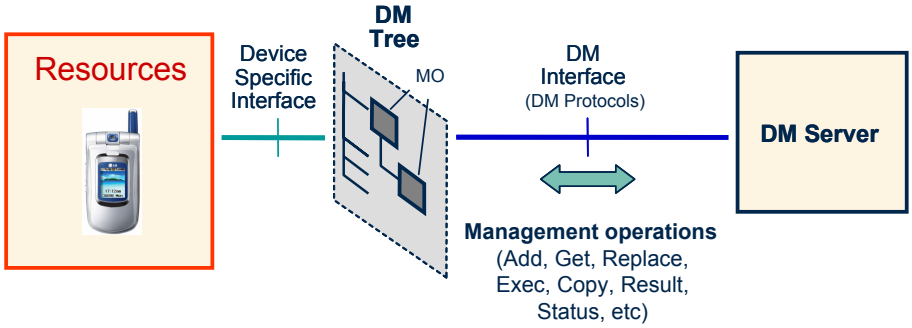


Fig. 2.18 – Architecture d'OMA DM

Les communications sont réalisées suivant le protocole OMA DM et qui utilise le langage SyncML (*Synchronization Markup Language*). SyncML est un langage basé sur XML qui définit les commandes d'administration nécessaires à la communication entre le client, *i.e.* l'appareil sans fil, et le serveur responsable de son administration.

Comme SNMP avec les MIB, OMA DM suit un modèle d'information hiérarchique appelé *DM tree* pour représenter ses ressources de manière uniforme. Un *DM tree* est composé de plusieurs *Management Objects* (MO). Un MO est un ensemble de paramètres qui ont des relations entre eux, comme les paramètres du constructeurs de l'appareil, les paramètres matériels ou les paramètres logiciels. La figure 2.19 montre un exemple de *DM tree*.

Dans l'architecture d'OMA DM, le serveur a la possibilité de modifier, de collecter, et de configurer ces paramètres à l'intérieur des différents MO. Cela se fait grâce au protocole DM et aux instructions d'administration qu'il définit et qui peuvent être exécutées sur les MO auxquels le serveur est autorisé d'accéder.

Firmware Update

Firmware Update Management Object (FUMO) [116] est une spécification en cours d'élaboration qui fournit des informations sur les MO associés au *firmware* dans OMA DM. Dans les appareils électroniques, le *firmware* représentait classiquement le microprogramme faisant le lien entre le matériel et le niveau assembleur. Aujourd'hui dans les appareils sans fil, le *firmware* englobe toutes les couches au dessus du matériel jusqu'au système d'exploitation inclus.

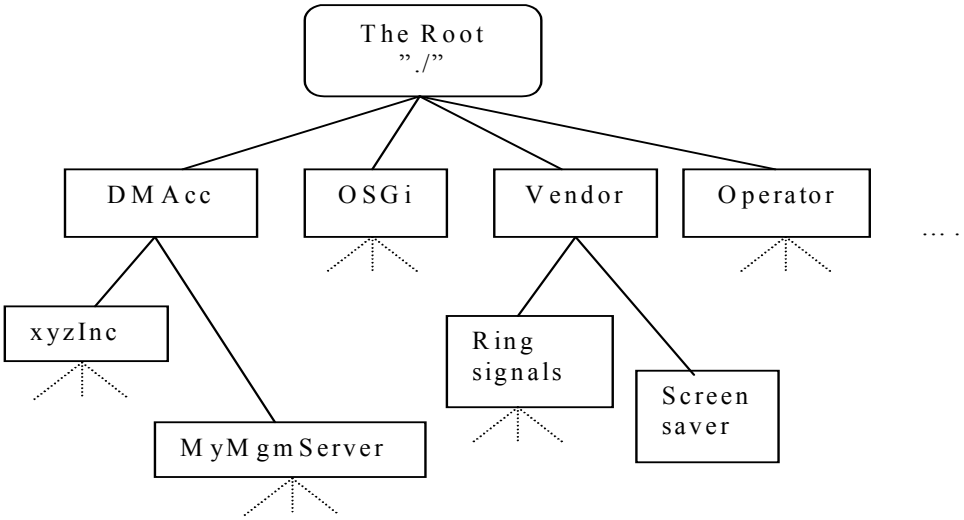


FIG. 2.19 – Exemple de *DM tree*

FUMO définit donc un MO permettant de mettre à jour le *firmware* par une communication sans fil avec le serveur. Un tel MO est représenté avec la description des nœuds qui le composent par la figure 2.20.

La mise à jour du *firmware* implique des échanges particuliers entre l'appareil sans fil et le serveur qui sont normalisés par la spécification. Le protocole de mise à jour est le suivant :

1. le serveur initie une session avec l'appareil,
2. l'appareil fournit les informations le concernant au serveur,
3. le serveur envoie le paquetage de mise à jour le plus approprié à l'appareil qui l'enregistre dans le nœud `x/Update/PkgData` du MO (opération `Replace`),
4. le serveur ordonne à l'appareil l'exécution de l'opération `Exec` pour rendre installer effectivement la mise à jour,
5. l'appareil confirme l'installation de la mise à jour.

Ce scénario est décrit par la figure 2.21.

Software Component Management

OMA DM propose le *Software Component Management Object* (SCOMO) [117] pour l'administration des composants logiciels embarqués dans les systèmes sans fil. A terme, le but de SCOMO est de permettre l'exécution à distance d'opérations d'administration sur les appareils sans fil. Les possibilités visées se limitent cependant à la gestion du cycle de vie des composants logiciels grâce à des opérations d'installation, de mise à jour, d'activation ou de désactivation des composants via un accès sans fil.

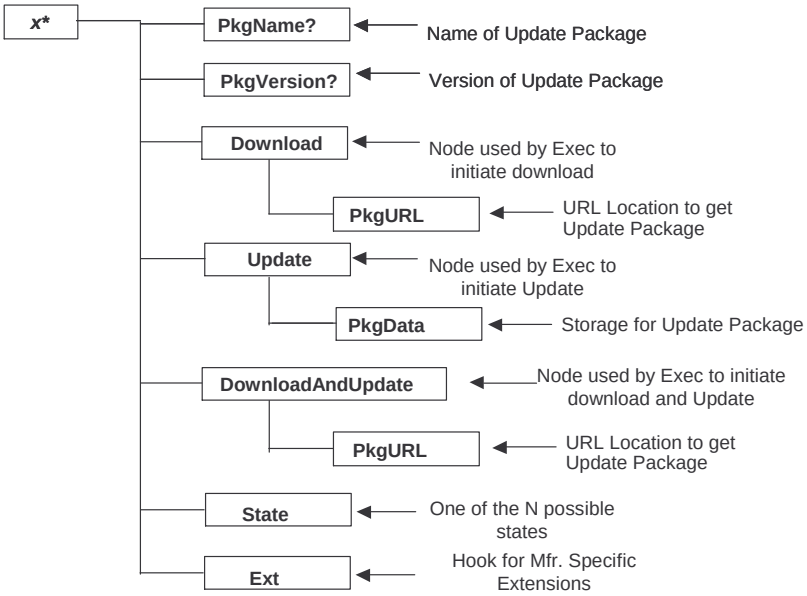


FIG. 2.20 – *OMA DM Firmware Update MO*

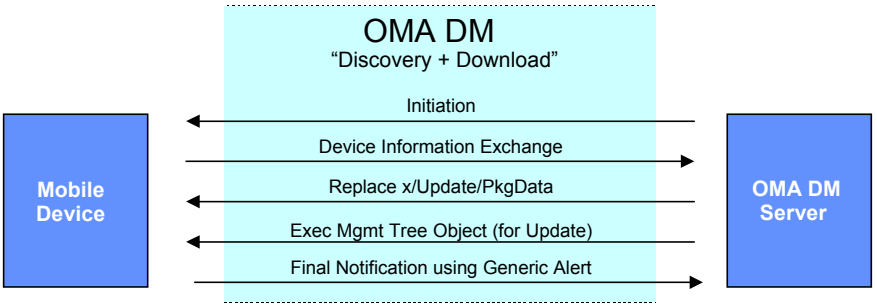


FIG. 2.21 – Exemple de commande *OMA DM Replace*

Dans la proposition actuelle, le cycle de vie classique des composants logiciels est étendu pour prendre en compte la livraison, c'est-à-dire le téléchargement des nouveaux composants. On peut constater ici un rapprochement vers l'approche orientée service où l'activité de livraison a une place prépondérante. La figure 2.22 représente le cycle de vie des composants dans SCOMO.

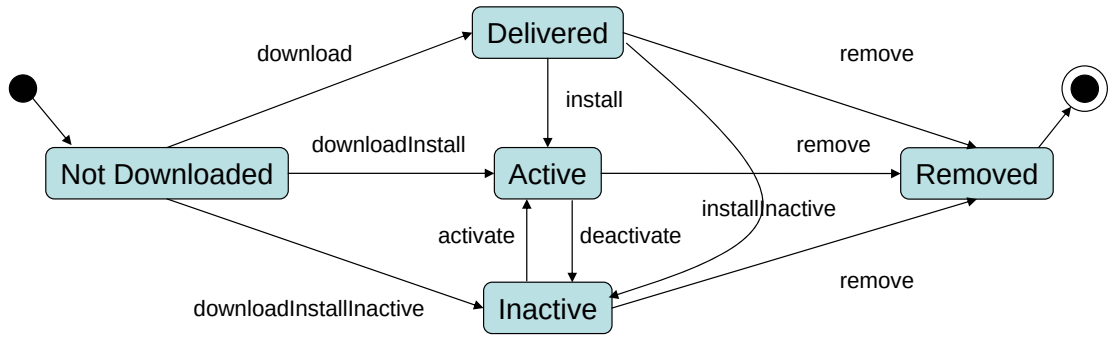


FIG. 2.22 – Cycle de vie des composants avec livraison dans SCOMO

Diagnostics and Monitoring

OMA DM propose également d'étendre sa spécification avec *DM Diagnostics and Monitoring* (DiagMon) [115] pour permettre à un système d'administration de détecter et de réparer des erreurs pouvant survenir sur un appareil sans fil.

DiagMon n'en est encore qu'à l'étape de l'analyse des besoins dans le processus de publication des standards d'OMA. Cette future technologie devrait donc prendre en compte :

- la gestion des politiques de diagnostics,
- le signalement des fautes,
- le suivi des performances,
- l'interrogation des appareils sans fil,
- l'invocation distante de procédures de diagnostic,
- la réparation distante des appareils sans fil.

2.4.2 JSR 232 & JSR 233

L'essor de la plateforme Java ME a naturellement amené la communauté Java à s'intéresser à l'administration des appareils sans fil embarquant cette technologie. Deux JSR (*Java Specification Requests*) ont été proposés pour faire évoluer la plateforme en ce sens : le JSR 232 intitulé *Mobile Operational Management* et le JSR 233 intitulé *J2EE Mobile Device Management and Monitoring Specification*.

Mobile Operational Management

Dans le cadre du JSR 232, la spécification de *Mobile Operational Management* [107] définit un paquetage optionnel de la plateforme Java ME. Son but est de spécifier une infrastructure d'administration pour les composants logiciels déployés sur des appareils supportant Java ME. Ce travail envisage l'administration au sein même des appareils sans fil en s'appuyant sur la plateforme de service OSGi. Les résultats correspondants ont d'ailleurs été inclus dans la dernière version d'OSGi, *Release* 4 [119].

Son modèle d'information, le *Mobile Management Tree*, est organisé sous forme arborescente de façon à assurer une compatibilité avec OMA DM. Le JSR 232 décrit donc ses propres MO pour uniformiser les informations d'administration concernant la configuration, la journalisation, la surveillance, les applications, le déploiement, et la sécurité.

La spécification décrit également la possibilité de construire des *Application Managers*, c'est-à-dire des *bundles* OSGi capables d'administrer des composants logiciels qui suivent un autre modèle technologique, comme MIDP par exemple. Pour cela, OSGi R4 propose un service d'administration qui uniformise l'accès aux ressources en proposant pour chaque composant applicatif un descripteur (*Application Descriptor*) pour la découverte du composant par son gestionnaire et une interface d'administration (*Application Handle*) pour la gestion du cycle de vie du composant. Cette architecture est représentée par la figure 2.23.

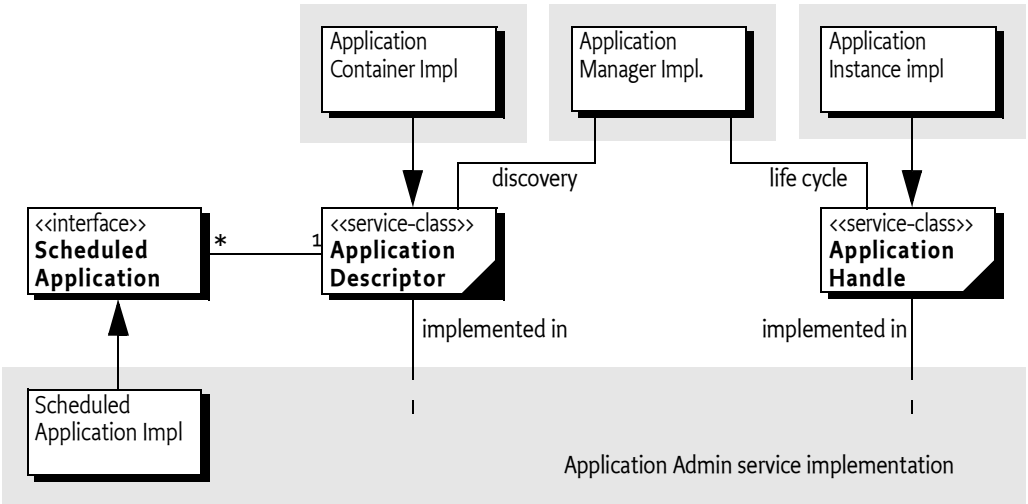


Fig. 2.23 – *Application Manager* dans le JSR 232

Ainsi ce JSR concerne principalement la gestion locale des composants logiciels mais ne définit pas d'infrastructure serveur pour une administration distante. Il repose sur le protocole OMA DM pour son utilisation avec des serveurs d'administration.

J2EE Mobile Device Management and Monitoring

Contrairement au *Mobile Operational Management* qui s'intéresse en particulier à l'administration côté appareil sans fil, le JSR 233 [148] concerne la construction des systèmes d'administration capables d'administrer à distance les composants logiciels déployés dans les systèmes sans fil.

Cette spécification prévoit de définir une extension de Java EE pour supporter l'administration des systèmes basés sur Java ME. Son développement, qui est encore en cours, portera en particulier sur la définition de fonctionnalités permettant aux systèmes d'administration :

- d'initier la livraison et l'installation à distance de nouveaux composants logiciels sur les systèmes sans fil,
- de mettre à jour, de supprimer ou de reconfigurer les composants logiciels déjà déployés sur les appareils sans fil,
- de fournir une vue de la configuration des composants logiciels comprenant en particulier un système de résolution des dépendances,
- de surveiller l'usage des applications sur les appareils sans fil.

2.4.3 Dynamically Programmable and Reconfigurable Software

Dynamically Programmable and Reconfigurable Software (DPRS) [134] est une approche permettant la construction d'un *middleware* reconfigurable pour les systèmes sans fil. DPRS met à profit une nouvelle technique, appelée *externalization*, pour augmenter les possibilités de contrôle des administrateurs sur les composants déployés. Le principe de l'*externalization* est d'extériorisser explicitement l'état, la logique et la structure interne des composants. Pour cela, les composants logiciels sont construits à partir des concepts de *Micro Building Blocks* (MBB), d'*Action* et de *Domain*.

Un MBB est la plus petite unité fonctionnelle adressable dans le système. La figure 2.24 représente son fonctionnement canonique :

1. il reçoit une collection de couples <*nom, valeur*> en paramètres d'entrée,
2. il exécute l'algorithme qu'il encapsule, ce qui peut modifier l'état de ses attributs dans la zone de stockage qui lui est allouée,
3. il génère une collection de couples <*nom, valeur*> en paramètres de sortie.

Les *actions* spécifient l'ordre d'exécution des MBB et par conséquent la logique des composants. Elles correspondent à un graphe orienté déterministe dont les nœuds sont des MBB marquant l'état d'exécution et les arêtes définissent l'ordre des transitions entre les MBB. A chaque transition est associée une expression conditionnelle portant sur les paramètres de sortie du MBB à l'origine de la transition afin de déterminer à l'exécution la prochaine transition et le prochain MBB à exécuter. 2.25

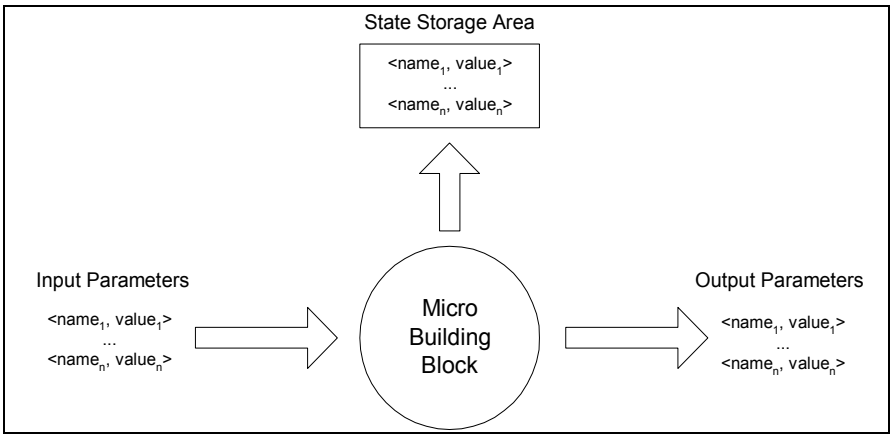


FIG. 2.24 – Micro Building Blocks [134]

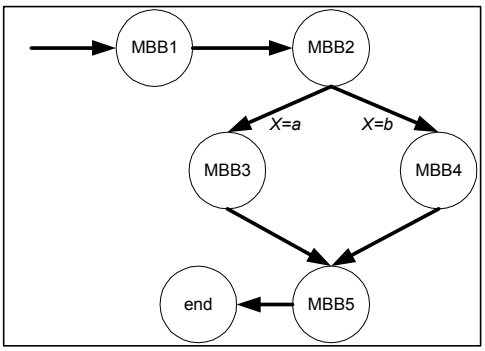


FIG. 2.25 – Action interprétée dans les composants DPRS [134]

Un *domain* correspond à un composant. Il regroupe une collection de MBB étroitement liés et fournit les zones de stockages nécessaires pour la structure du composant (*i.e.* la liste des MBB), sa logique (*i.e.* la liste des actions) et son état (*i.e.* les attributs des MBB et l'état d'exécution). La figure 2.26 représente l'organisation d'un tel composant.

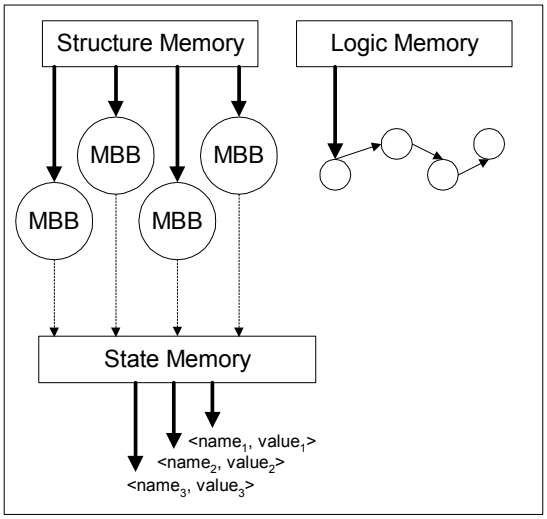


FIG. 2.26 – Externalisation dans les composants DPRS [134]

DPRS repose sur une infrastructure qui permet d'administrer les composants à l'exécution. Pour ce faire, cette infrastructure s'appuie sur deux composants principaux : le *Domain Manager* et le *MBB Scheduler*. Le *Domain Manager* fournit les services permettant de modifier à l'exécution :

- les fonctionnalités des composants en remplaçant leurs MBB,
- le comportement des composants en modifiant les transitions des actions.

Le *MBB Scheduler* maintient et exporte des informations sur l'exécution du système, ce qui comprend :

- les actions en cours d'exécution,
- les MBB en cours d'exécution,
- les paramètres associés aux actions.

Grâce à sa technique d'*externalization*, DPRS permet ainsi une administration précise du comportement à l'exécution des composants logiciels. Cette approche ne permet cependant pas de contrôler la validité des reconfigurations opérées à l'exécution (cohérences des nouvelles transitions). Sa principale contribution est de permettre de déterminer des points de reconfiguration sûrs : l'exécution atomique des MBB définit des étapes dans le comportement des composants logiciels, à la fin desquelles les modifications peuvent être effectuées sûrement vis-à-vis de l'exécution globale en cours.

DPRS se concentre sur son infrastructure locale et ne définit pas les mécanismes d'accès au *Domain Manager* pour administrer à distance les composants déployés dans le système sans fil. En outre, l'évaluation proposée dans [134] s'appuie sur des tests effectués sur des systèmes utilisant des processeurs performants (Pentium IV et Pentium M), ce qui montre que cette approche est encore en phase expérimentale dans le domaine du sans fil.

2.5 Synthèse

Dans ce chapitre, nous avons tout d'abord présenté les concepts et principes généraux de l'administration à travers une étude des premiers standards d'administration qui ont été définis pour les besoins de la gestion des réseaux et de leurs équipements. L'évolution de ces approches a eu pour but d'une part de résoudre les problèmes de passage à l'échelle en adoptant une architecture centralisée puis une architecture hiérarchique, et d'autre part d'automatiser le processus d'administration, voire même de conférer une certaine autonomie aux systèmes d'administration, avec les approches par politique et les approches autogérées.

Lors de cette étude, nous avons pu constater que les activités d'administration définies originellement pour les réseaux et les télécommunications n'étaient pas optimales pour administrer au mieux les applications logicielles. Nous avons ainsi caractérisé l'administration des applications par la surveillance et le contrôle de trois types d'états particulièrement importants à administrer sur le logiciel : l'état de cycle de vie, l'état de configuration et l'état de fonctionnement. Les travaux que nous présentons dans cette thèse abordent spécifiquement l'administration de l'état de fonctionnement des composants logiciels déployés sur des systèmes sans fil.

En section 2.3, nous avons dressé l'état de l'art des approches d'administration pour les composants logiciels dans le cadre général selon la taxonomie suivante :

- les approches basées sur l'infrastructure qui sont principalement utilisées pour administrer l'état de cycle de vie des composants au sein de leur infrastructure mais fournissent également des solutions de bas niveau pour administrer les deux autres types d'état,
- les approches basées sur l'architecture qui administrent l'état de configuration des applications à base de composants et indirectement l'état de fonctionnement,
- les approches basées sur le comportement qui administrent spécifiquement l'état de fonctionnement des composants.

Au niveau du contrôle de l'état de fonctionnement, les approches basées sur l'architecture s'appuient toutes sur la même hypothèse : « il existe un composant qui corrige le problème ». Or ce n'est que rarement le cas, à moins que le composant correctif soit déjà envisagé au développement de l'application. Ou alors faut-il attendre une nouvelle version du composant défectueux ? Mais comment le corriger ? Et sur quelles bases ?

Les approches basées sur le comportement des composants utilisent des modèles pour répondre à ces questions. Ces approches s'appuient sur les entrées et sorties des composants et

les comparent à leur comportement supposé. Cependant le caractère « boîte noire » des composants logiciels empêche leur mécanique interne d'être totalement perçue seulement à partir de l'observation de leurs modifications extérieures et ne permet donc pas un réel contrôle de l'application.

Enfin en section 2.4, nous avons fait état du peu de solutions actuellement disponibles pour l'administration des composants dans le domaine du sans fil. Les technologies comme JMX ne sont pas disponibles dans ces environnements mais on remarque un certain engouement industriel pour établir de nouveaux standards d'administration spécifiques au sans fil. Influencées par l'émergence des approches orientées service, ces solutions s'orientent plus prioritairement sur l'administration de l'état de cycle de vie des composants et en particulier sur leur livraison. DPRS est l'une des rares approches à s'attaquer à l'administration de l'état de fonctionnement des composants. Mais si l'approche est conceptuellement intéressante, ses expérimentations en environnement non contraint traduisent la difficulté de réaliser un véritable système d'administration déployable sur des systèmes sans fil ayant des ressources limitées par rapport aux stations de travail pour lesquelles des solutions existent.

Ceci conclut la première partie de notre dissertation. Maintenant que nous avons posé la problématique de nos travaux et exploré les solutions existantes, nous pouvons présenter dans les chapitres suivants notre contribution à l'administration de l'état de fonctionnement des composants logiciels déployés sur des systèmes sans fil.

Chapitre 3

Conception de composants logiciels en vue de leur administration

Une approche basée sur le comportement adaptée aux systèmes sans fil

Sommaire

3.1 Présentation générale de l'approche 74

3.2 Modélisation de composants UML 2 pour les systèmes sans fil . . . 75

3.2.1 Introduction à UML 2 77

3.2.2 Structure : modèle de composants UML 2 79

3.2.3 Comportement : machines à états UML 2 82

3.3 PauWare.Velcro : exécutabilité et persistance à l'exécution de machines à états UML 2 87

3.3.1 MDA et Executable UML 87

3.3.2 Principes de PauWare.Velcro 88

3.3.3 Principes du modèle de composants PauWare 95

3.4 Synthèse 100

Dans le premier chapitre de ce manuscrit, nous avons introduit l'ingénierie logicielle basée composants et son application au domaine du sans fil, puis dans le deuxième chapitre nous avons dressé un état de l'art des différentes solutions d'administration existantes, en particulier pour les systèmes à base de composants.

Dans ce chapitre, nous présentons maintenant une approche de conception qui permet de réifier le comportement des composants logiciels et dont tire parti l'architecture d'administration sans fil que nous avons conçue et que nous décrirons dans les chapitres suivants.

3.1 Présentation générale de l'approche

Notre proposition vise à améliorer l'administration des composants logiciels dans le contexte d'un déploiement dans des systèmes sans fil. Pour cela nous voulons que la prise en compte du comportement à l'exécution des composants soit réellement possible dans l'activité d'administration. En d'autres termes, notre objectif est de permettre à la fois l'observation et le contrôle de l'état des composants, et plus précisément de leur état de fonctionnement tel que nous l'avons défini dans le chapitre précédent (cf. 2.1.3 p. 39). Nous proposons donc une nouvelle approche d'administration à la fois basée sur le comportement des composants logiciels et adaptée aux systèmes sans fil. La figure 3.2 schématise les différentes étapes de notre approche pour aboutir à un composant administrable.

Le principe de notre approche est d'exprimer le comportement des composants logiciels sous la forme d'un modèle exploitable durant les différentes phases de développement et de le faire persister à l'exécution. Défini lors de la modélisation, le comportement enfoui dans les composants est matérialisé et exécuté par un gestionnaire interne de sorte que les composants puissent être manipulés pour les besoins de l'administration. Cela peut s'apparenter à la façon dont les approches basées sur l'architecture permettent que l'architecture spécifiée avec un ADL dans les premières phases du développement du logiciel soit rendue accessible à l'exécution (cf. 2.3.2 p. 54).

De même, notre approche peut être comparée aux approches basées sur le comportement que nous avons étudiées dans le chapitre 2 (cf. 2.3.3 p. 58), puisque, à l'instar de celles-ci, nous utilisons des modèles pour spécifier le comportement des composants. Par contre, nous prenons en compte les besoins de l'administration beaucoup plus tôt dans le cycle de vie des composants, ce qui nous permet de les rendre réellement administrables. Les approches existantes qui sont également basées sur le comportement, n'envisagent l'administration qu'après le développement des composants. Elles se heurtent ainsi à l'aspect « boîte noire » des composants, ce qui limite ces approches à l'observation externe du comportement des composants seulement.

Afin que notre approche soit applicable aux systèmes sans fil[7], nous avons choisi un modèle éprouvé dans ce domaine [59, 45, 95] : les machines à états d'UML 2 issues des *Statecharts* de Harel [62]. Le choix d'UML 2 nous permet également de lier le comportement à la structure des composants grâce aux *Component Diagrams* et aux *Composite Structure Diagrams*, sans devoir se limiter à un modèle[8] de composants « technologique » particulier.

Pour l'administration, l'avantage de l'utilisation des machines à états réside dans leur pouvoir d'abstraction. En effet, dans un système d'administration classique, la surveillance d'une variable d'un programme administré s'effectue à travers l'observation continue des variations de valeurs de la variable sur l'ensemble des valeurs définies par son type. Les machines à états, elles,

[7]Les aspects spécifiques au domaine du sans fil sont plutôt d'ordre architectural ou technique comme nous le verrons dans les chapitres suivants. L'approche présentée ici est adaptée aux systèmes sans fil mais n'est pas exclusive à ce type de systèmes.

[8]Ici le terme « modèle » a plus le sens de « format » que de représentation issue d'une modélisation.

permettent de capturer seulement les variations qui ont une signification pertinente au niveau du comportement du programme. Ces variations sont discrétisées en états abstraits du programme. Ce sont ces états abstraits que nous exposons dans notre système d'administration.

Ainsi, une variable `i` de type entier pourra par exemple être considérée au niveau du comportement du programme à travers les trois états `Négatif`, `Zéro` et `Positif`. La machine à états de la figure 3.1 montre un exemple de comportement basé sur ces trois états. Les invariants d'états explicitent ici les relations entre les états concrets, exprimés par les valeurs de la variable `i`, et les états abstraits, exprimés par les états de l'automate.

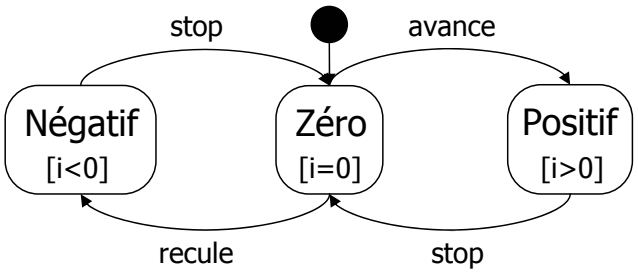


Fig. 3.1 – Exemple de comportement utilisant des états abstraits de la variable `i`

L'exécutabilité et la persistance à l'exécution des machines à états sont assurées par la technologie PauWare, à laquelle nous avons contribué et dont nous avons réalisé la version PauWare.Velcro pour les systèmes sans fil. De façon à assurer une correspondance directe entre le modèle et le code, PauWare, et par conséquent PauWare.Velcro, fournissent les constructions nécessaires pour implémenter un comportement modélisé à partir de *Statecharts*.

Le comportement de nos composants administrables est ainsi réalisé par une machine à états PauWare. Celle-ci est placée sous l'autorité d'un gestionnaire interne qui est responsable du contrôle de l'exécution de la partie fonctionnelle interne des composants. En communiquant avec le gestionnaire interne d'un composant administrable, le système d'administration global peut alors observer et contrôler le comportement du composant en manipulant directement sa machine à états.

Dans la suite de ce chapitre, nous présentons les modèles UML 2 que nous utilisons pour spécifier la structure des composants et leur comportement (section 3.2). Puis, nous décrivons les principes de PauWare.Velcro pour l'exécutabilité et la persistance des modèles (section 3.3). Enfin, nous concluons par une synthèse du chapitre (section 3.4).

3.2 Modélisation de composants UML 2 pour les systèmes sans fil

Dans cette section, nous présentons une brève introduction à UML 2. Ensuite, nous nous intéressons plus particulièrement aux *Component Diagrams* et aux *Composite Structure Diagrams*

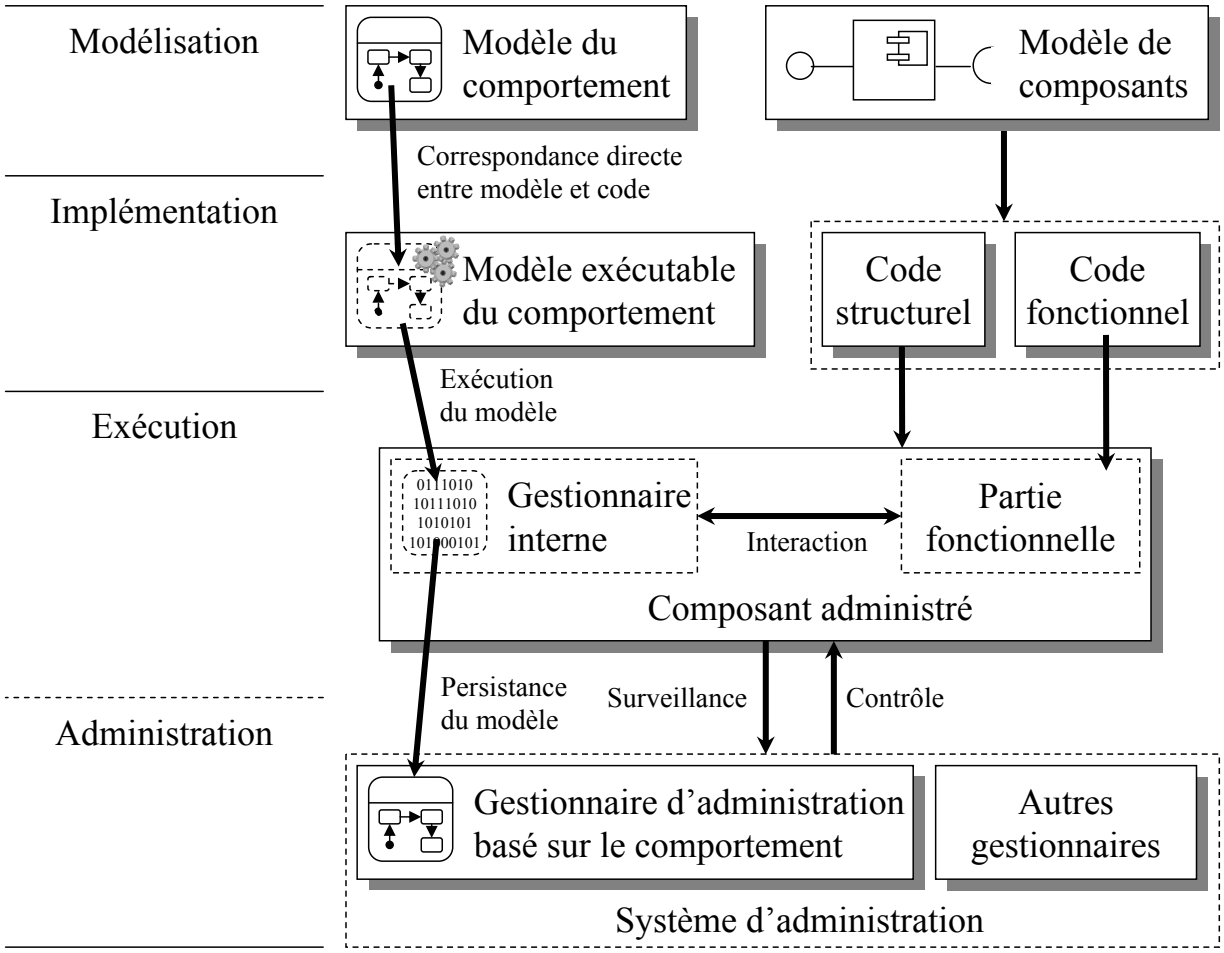


FIG. 3.2 – De la modélisation à l'administration : conception d'un composant logiciel administrable

que nous utilisons pour modéliser la structure de nos composants, puis aux *State Machine Diagrams* que nous utilisons pour leur comportement.

3.2.1 Introduction à UML 2

Le langage de modélisation graphique UML (*Unified Modeling Language*) [113] est devenu le standard *de facto* pour modéliser les systèmes logiciels. Son utilisation permet de visualiser, de spécifier, de construire et de documenter les différents aspects et artefacts de ces systèmes [19].

Son origine provient de la fusion de trois langages de modélisation orientée-objet, issus des méthodes de Grady Booch (*Booch Method*), James Rumbaugh (*Object Modeling Technique*) et Ivar Jacobson (*Objectory*). La première version officielle d'UML fut acceptée par l'OMG (*Object Management Group*) en 1997 et publiée en tant que UML 1.1. La norme s'est depuis enrichie et a évolué de version en version pour aboutir en 2007 à son actuelle version 2.1.1.

La spécification d'UML définit une notation et un métamodèle. La notation correspond aux éléments graphiques que l'on peut voir dans les modèles. Elle correspond à la syntaxe concrète d'UML qui couvre un large spectre de langages graphiques différents. Le métamodèle, lui, permet de décrire assez rigoureusement les concepts d'UML et correspond à la syntaxe abstraite du langage global.

Un ensemble de notations

Pour être suffisamment complète, la modélisation d'un système doit permettre de comprendre le système selon différents points de vue. Dans un but d'universalité, UML englobe un ensemble de notations variées et adaptées à la description de différentes vues d'un système à modéliser. Ces vues sont représentées graphiquement à l'aide de diagrammes.

Il existe 13 types de diagrammes dans UML 2 que l'on peut regrouper en 3 catégories :

- Les diagrammes structurels permettent de représenter la structure d'un système, autrement dit, ses aspects statiques. Parmi ces diagrammes, on trouve :
  - le diagramme de classes (*Class Diagram*) qui expriment la structure statique du système en termes de classes et de relations entre ces classes (association, composition, agrégation, héritage, dépendance, ...),
  - le diagramme d'objets (*Object Diagram*) qui représente les instances de classes utilisées dans le système ainsi que leurs liens,
  - le diagramme de déploiement (*Deployment Diagram*) qui permet de représenter la configuration des différents matériels du système, appelés nœuds, et la répartition des composants logiciels sur ces nœuds,
  - le diagramme des paquetages (*Package Diagram*) qui sert à organiser en regroupements logiques les différents modèles qui représentent un même système,

  - le diagramme de composants (*Component Diagram*) qui permet de structurer le système en termes de composants logiciels selon les principes de l'ingénierie logicielle basée composant,
  - le diagramme de structures composites (*Composite Structure Diagram*) qui permet de décomposer hiérarchiquement un classifieur UML (pas seulement un composant) en décrivant sa structure interne et de décrire des collaborations[9].

- Les diagrammes comportementaux sont utilisés pour représenter le comportement du système, sa dynamique. Les diagrammes comportementaux comprennent :
  - le diagramme des cas d'utilisation (*Use Case Diagram*) qui décrit les grandes fonctionnalités du système du point de vue de l'utilisateur et des besoins, et correspond donc plus à de l'ingénierie des besoins qu'à de la modélisation [9],
  - le diagramme d'activité (*Activity Diagram*) qui représente une vision fonctionnelle du système en termes d'actions et d'activités suivant une sémantique proche des Workflow (niveau haut) et de SDL (niveau bas),
  - le diagramme de machine à états (*State Machine Diagram*) qui permet de décrire le comportement du système en termes d'états et de transitions sous forme de machine à états finis.

- Les diagrammes d'interactions sont un sous-ensemble des diagrammes comportementaux. Ils permettent de représenter les flots de contrôle et de données entre les différents objets du système modélisé. On dénombre quatre types de diagrammes d'interactions :
  - le diagramme de séquence (*Sequence Diagram*) qui permet de décrire les interactions entre les objets en mettant en avant l'ordonnancement temporel des messages échangés,
  - le diagramme de communication (*Communication Diagram*), qui, comme le diagramme de séquence, décrit les échanges de messages au sein du système mais en se concentrant sur les dépendances des objets qui envoient et reçoivent des messages[10]. Les diagrammes de communication et de séquence sont deux vues différentes mais logiquement équivalentes,
  - le diagramme global d'interaction (*Interaction Overview Diagram*) qui est une variante du diagramme d'activité permettant de décrire les enchaînements possibles entre les scénarios préalablement identifiés sous forme de diagrammes de séquences,
  - le diagramme de temps (*UML Timing Diagram*) qui permet de définir des contraintes temporelles sur les objets.

[9] La partie structurelle de l'ancien diagramme de collaboration (*Collaboration Diagram*) d'UML 1 est maintenant intégrée dans le diagramme de structures composites

[10] Cela correspond à la partie dynamique de l'ancien diagramme de collaboration (*Collaboration Diagram*) d'UML 1.

Métamodélisation

La contrepartie de la grande expressivité des langages de modélisation graphiques est souvent leur manque de rigueur. A l'inverse dans le domaine des méthodes formelles, les langages de spécification dérivent de *calculi* mathématiques et n'offrent pas de place à l'ambiguïté quant à leur définition. Afin de rendre explicite et plus solide la sémantique de sa notation, la syntaxe abstraite d'UML est définie par un métamodèle.

A l'origine, UML était décrit en UML. Un sous-ensemble d'UML était utilisé pour définir les constructions et les règles du langage lui-même, comme cela peut se faire dans d'autres langages réflexifs. Le besoin d'exprimer des métamodèles pour d'autres langages de modélisation a conduit l'OMG à standardiser la partie réflexive d'UML pour définir un langage dédié à la définition de métamodèles, autrement dit un métamétamodèle : le MOF (*Meta Object Facility*).

Les fondations d'UML ont ainsi été rebâties autour du MOF dans une architecture de langages de modélisation à quatre niveaux. La figure 3.3 montre un exemple des relations entre les concepts des différents niveaux. Chaque niveau décrit les éléments utilisés par le niveau inférieur et le niveau M0 représente le monde réel. Réciproquement, chaque élément d'un même niveau doit se conformer à un élément de modélisation du niveau supérieur excepté pour le dernier niveau M3 qui est réflexif et se conforme à lui-même.

Ainsi, le MOF constitue le niveau M3 et permet de décrire au niveau M2 le métamodéle d'UML mais également le métamodèle d'autres langages comme SPEM (*Software Process Engineering Metamodel*) ou EDOC (*Enterprise Distributed Object Computing*) entre autres. Au niveau M2, le métamodèle d'UML définit par exemple les concepts d'attribut, de classe et d'instance qui seront utilisés au niveau M1 par les utilisateurs d'UML dans leurs modèles. Enfin, au niveau M1, un modèle UML constitue une abstraction du système à modéliser et de ses instances à l'exécution qui correspondent au niveau M0.

3.2.2 Structure : modèle de composants UML 2

La modélisation des composants avec UML a considérablement évolué entre UML 1 et UML 2. Outre le changement de notation des *Component Diagrams*, cette évolution se traduit dans le niveau de compréhension du concept de composant lui-même. Dans les premières versions d'UML 1, les composants étaient considérés au niveau implémentation et ils représentaient seulement des éléments physiques (code source et implémentation à l'exécution). Le passage de UML 1 à UML 2 a été fortement influencé par l'ingénierie logicielle basée composant et cela se retrouve dans la nouvelle version.

Component Diagrams

Dans UML 2, les composants ne sont plus seulement pris en compte au niveau implémentation mais également au niveau conceptuel avec la possibilité de définir des « composants logiques » :

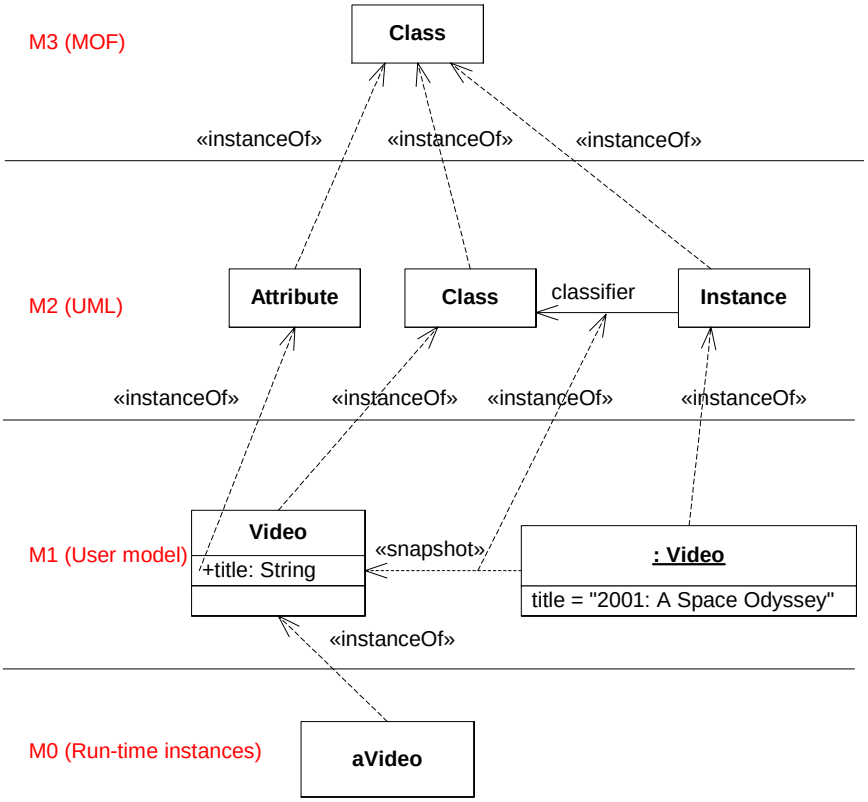


FIG. 3.3 – Exemple des quatre niveaux de modélisation d'UML (extrait de [112])[a]

[a]Bien qu'extrait de la spécification d'UML 2, cet exemple ne satisfait pas le métamodèle d'UML 2 puisque la métaclasse *Instance* au niveau M2 est maintenant remplacée par celle d'*InstanceSpecification*. Cet exemple permet néanmoins de visualiser les relations entre les niveaux de modélisation dans UML.

> *« The Components package supports the specification of both logical components (e.g., business components, process components) and physical components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.) [...]. »* [113]

Nous avons déjà introduit la notation des diagrammes de composants UML 2 pour illustrer les concepts relatifs à l'ingénierie des composants (cf. figures 1.1, 1.2, 1.3, 1.4 et 1.5 pages 7, 8, 9 et 17). On retrouve ainsi ces mêmes concepts au niveau du métamodèle d'UML à travers la métaclasse `Component`, représentée en figure 3.4.

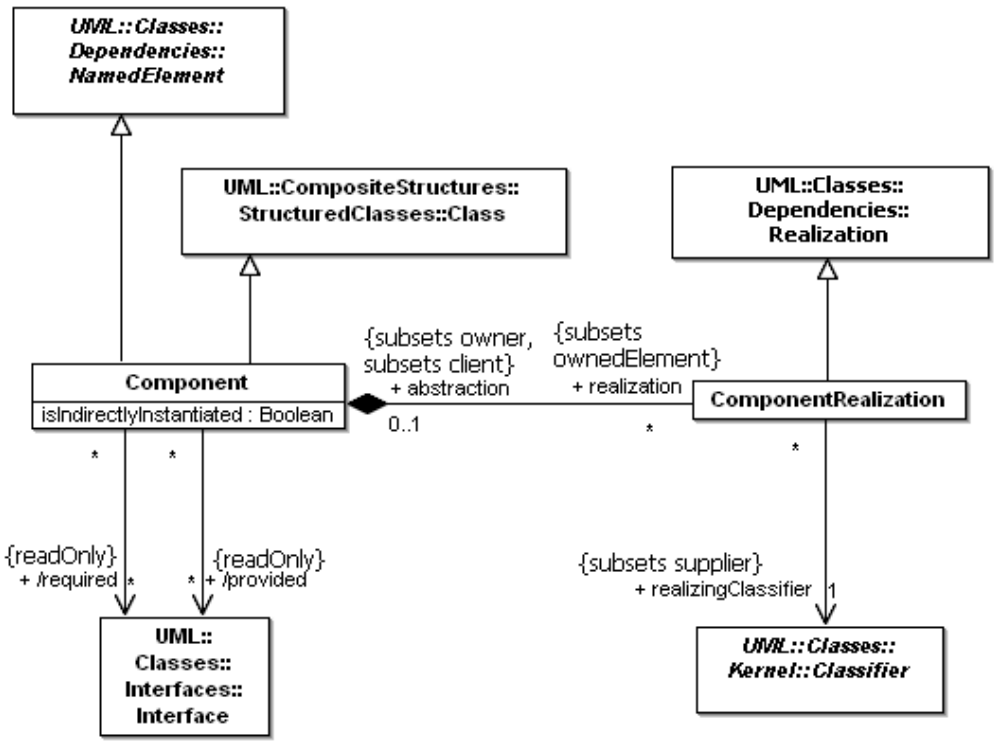


Fig. 3.4 – Métamodèle des composants UML (extrait de [113])

Un composant est donc défini comme une spécialisation d'une classe qui peut être réalisée par plusieurs autres classifieurs (une composition de `ComponentRealisation`) et qui comporte une spécification externe décrite par une ou plusieurs interfaces. Ces interfaces, qui peuvent être fournies (association dérivée `/provided`) ou requises (association dérivée `/required`), constituent les bases permettant de connecter les composants entre eux.

UML propose deux types de connecteurs pour les composants :

- le connecteur d'assemblage (*Assembly Connector*) qui permet de relier deux composants dont un fournit les services que le second requiert. Ce connecteur associe une interface de services fournis à une interface de services requis. Cela correspond à une composition horizontale telle que nous l'avons définie au premier chapitre (cf. figure 1.2),

- le connecteur de délégation (*Delegation Connector*) qui lie la structure externe d'un composant à sa structure interne en délégant la charge du contrat spécifié par les interfaces du composant aux constructions internes qui le composent. Ce connecteur associe deux interfaces de services fournis ou deux interfaces de services requis. Cela correspond à une composition verticale telle que nous l'avons définie au premier chapitre (cf. figure 1.3).

Les *Composite Structure Diagrams* permettent de décrire plus en détails ces constructions internes.

Composite Structure Diagrams

Nous avons vu précédemment qu'un composant est défini dans le métamodèle d'UML comme une spécialisation du type `Class`, mais pas n'importe lequel. En effet, il existe plusieurs variantes du type `Class` dans le métamodèle, et la variante en question raffine le concept pour les *Composite Structure Diagrams* (paquetage `UML::CompositeStructures`). Ces diagrammes ajoutent la possibilité aux classifieurs de posséder une structure interne (paquetage `UML::CompositeStructures::InternalStructures`) :

> *« The InternalStructure*[11] *subpackage provides mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier. A structure of this type represents a decomposition of that classifier and is referred to as its "internal structure." »* [113]

Par héritage, cette propriété est donc également conférée aux composants. Il est ainsi possible de décrire un composant comme étant la composition d'autres composants de granularité inférieure : on définit ainsi un composant composite. La figure 1.3 de la page 8 montre un exemple d'une telle construction.

On retrouve ici la notion de composition verticale ou hiérarchique notablement promulguée par les ADL. A travers l'adoption de ces nouveaux diagrammes dans sa version 2, on voit comment UML continue de s'inspirer de nombreuses autres notations, comme cela est le cas depuis ses débuts avec par exemple l'intégration des *Statecharts* pour la spécification de comportements.

3.2.3 Comportement : machines à états UML 2

Les machines à états UML 2, ou plus exactement les *behavioral state machines*[12], sont des diagrammes de comportement basés sur les *Statecharts* de Harel [62]. Les *Statecharts* étendent les mécanismes des automates finis conventionnels par l'ajout des notions de hiérarchie, de concurrence et de communication, ce qui peut être résumé simplement par l'égalité suivante :

[11] (*sic*) Le nom du paquetage comporte un « s » final partout ailleurs dans la documentation d'UML.

[12] UML 2 distingue deux types de machines à états : les *behavioral state machines*, que nous utilisons, et les *protocol state machines* qui sont des machines à états allégées (pas de possibilité de spécifier des activités dans les états par exemple), prévues pour décrire des protocoles ou des scénarios d'utilisation de classifieurs UML particuliers comme les interfaces ou les ports. Nous ne traitons pas ce dernier type de machines à états dans cette thèse.

« statecharts = state-diagrams + depth + orthogonality + broadcast-communication. » [62]

Automates finis

Les automates finis constituent un modèle mathématique éprouvé et fréquemment employé en informatique, notamment pour l'analyse des langages. Leur utilisation à travers leurs diagrammes correspondants peut également permettre, dans certaines limites, de modéliser le comportement dynamique des systèmes réactifs. On représente un automate fini par un diagramme d'états-transitions qui correspond à un graphe orienté étiqueté dont les sommets sont les états de l'automate et dont les arêtes étiquetées sont les transitions.

Dans les *Statecharts*, une transition traduit un changement d'état dans le comportement du système qui s'opère sous certaines conditions et en réponse à un événement reçu. La figure 3.5 représente la forme canonique d'une transition dans un diagramme d'états-transitions avec gardes.

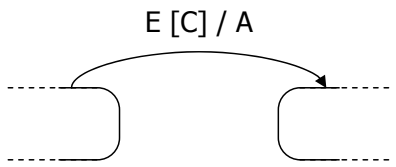


Fig. 3.5 – Forme canonique d'une transition dans un diagramme d'états-transitions

Si l'état courant de l'automate est l'état d'origine de la transition quand l'événement E est reçu, alors si la condition C est vérifiée, l'action A est exécutée et l'état courant de l'automate devient l'état cible de la transition. On retrouve ici le formalisme ECA [35, 43] auquel est ajouté la notion d'état. Cette notion est importante d'un point de vue modélisation puisqu'elle permet de capturer les différents moments clés du fonctionnement du système afin de pouvoir raisonner sur celui-ci.

La figure 3.6 représente le comportement élémentaire d'un système. Le comportement n'est décrit que par deux états, `On` quand le système est en fonctionnement et `Off` quand le système est arrêté. On peut déjà apprécier sur cet exemple simple l'expressivité apportée par les automates aux règles ECA. En effet, les règles portées pas les transitions de cette machine à états sont de type E (*Event* seulement) alors que dans des systèmes classiques utilisant le formalisme ECA, les règles doivent être de types EA (*Event-Action*), CA (*Condition-Action*) ou ECA pour décrire un comportement.

Bien qu'une action concrétise effectivement l'évolution d'un comportement, les diagrammes d'états-transitions permettent de spécifier une modification de comportement par un simple changement d'état. Même si aucune action n'est associée à la transition provoquant ce changement et qu'aucune activité n'est associée au nouvel état, le seul fait de changer d'état a une incidence sur les événements auxquels réagira le système, et par extension sur son comportement.

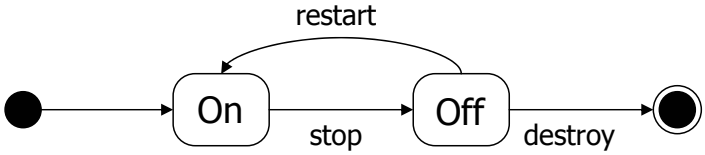


FIG. 3.6 – Exemple de machine à états simple

Hiérarchie

Les *Statecharts* possèdent la notion d'états hiérarchiques. Cela permet de définir le comportement d'un système selon plusieurs niveaux d'abstraction en décomposant un état complexe en sous-états plus précis.

Ainsi, l'exemple de la figure 3.6 est général et peut s'appliquer à de nombreux systèmes. Il peut être raffiné pour spécifier par exemple le comportement d'un serveur quelconque traitant une seule requête à la fois. La figure 3.7 représente un diagramme d'états-transitions utilisant des états hiérarchiques pour modéliser un tel comportement.

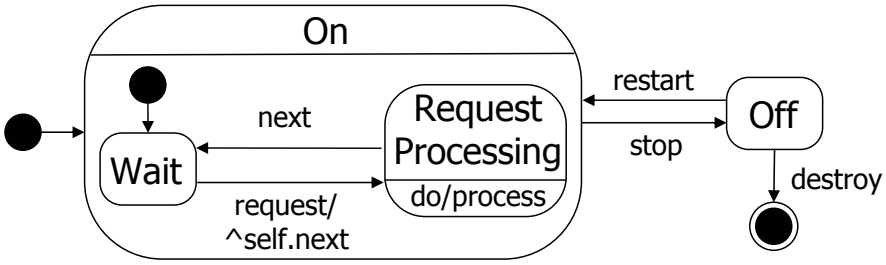


FIG. 3.7 – Exemple de machine à états avec avec états hiérarchiques

Dans le cas des automates avec états hiérarchiques, plusieurs états peuvent être actifs simultanément. Dans l'exemple, si l'état `Wait` est actif cela implique que l'état `On` l'est aussi. De même, si l'état `Request Processing` est actif alors l'état `On` l'est également.

Un comportement décrit par un automate hiérarchique peut l'être également par un automate classique. L'utilisation des automates hiérarchiques apporte cependant des avantages qui sont : une meilleure intelligibilité des modèles et une diminution importante du nombre d'états et de transitions nécessaires dans la modélisation de comportements complexes.

Concurrence

La notion de hiérarchie que nous venons d'introduire permet de décomposer les états en sous-états mutuellement exclusifs, ce qui est le mécanisme de base des automates. Cette décomposition est qualifiée de décomposition XOR (*exclusive or*). Le formalisme des *Statecharts* supporte également une décomposition AND des états. Cette décomposition permet de traduire le fait

que, étant dans un état, le système se trouve à la fois dans plusieurs des sous-états de cet état composite. La notation utilisée est une ligne pointillée séparant les états orthogonaux.

Dans notre dernier exemple en figure 3.7, le serveur modélisé ne pouvait gérer qu'une seule requête à la fois : les requêtes passées lors du traitement d'une autre étaient perdues. La figure 3.8 décrit le comportement d'un serveur capable de gérer plusieurs requêtes. Ce comportement utilise une décomposition AND de l'état `On` pour permettre d'enregistrer les différentes requêtes à l'aide d'une file (état `Listen`) et tout en continuant de traiter simultanément la requête en cours (état `Request Processing`).

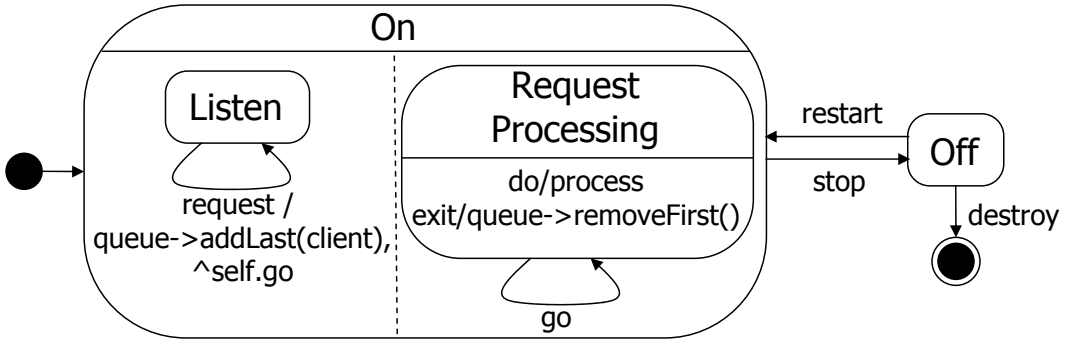


Fig. 3.8 – Exemple de machine à états avec états orthogonaux

Tout en améliorant l'intelligibilité des diagrammes, ce mécanisme permet également de faire face à l'explosion du nombres d'états à laquelle sont confrontés les automates classiques dans la modélisation de comportements complexes.

Communication asynchrone

D'un point de vue architectural, la décomposition d'un système en un assemblage de composants permet d'obtenir une structure intelligible du système. De la même manière, la modélisation du comportement d'un système complet étant difficile à appréhender de façon monolithique, il est préférable de décrire individuellement le comportement des différentes parties du système. Dans le cadre d'une modélisation basée sur des *Statecharts*, il est alors intéressant de pouvoir faire communiquer les machines à états de ces différentes parties.

A la différence d'autres langages de modélisation, le formalisme des Statecharts repose sur une communication *broadcast* :

> *« [...] there are some interesting differences between languages like CCS or Hoare's CSP [...] and the statechart formalism. [...] In these languages, the basic communication mechanism is a rendezvous, which enforces synchronization between a single sender and a single receiver. The statechart communication mechanism, on the other hand, is based on broadcast, whereby the sender may proceed even if nobody is listening. »* [62]

Ainsi, la communication de ces machines à états fonctionne par la diffusion de messages asynchrones. Lors de la diffusion, l'émetteur ne se préoccupe pas de l'état du receveur. Dans les machines à états UML 2, cette communication est décrite par un envoi de signal caractérisé par le symbole ^ et placé dans la partie *Action* d'une transition. La figure 3.9 montre la forme canonique d'une telle transition. Le signal envoyé doit correspondre à un événement de la machine à états du receveur qui y réagira en conséquence.

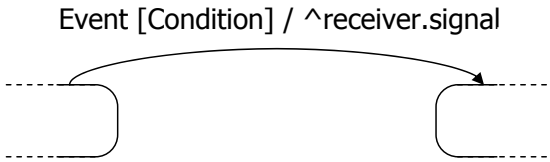


FIG. 3.9 – Forme canonique d'une transition générant un signal dans un *Statechart*

Traitement des événements : cycles *run-to-completion*

A l'exécution, une machine à états alterne entre deux modes : un mode passif, durant lequel elle est en attente de l'arrivée du prochain événement, et un mode actif, pendant lequel elle traite un événement. Comme les communications suivent un modèle asynchrone, un événement peut être reçu par une machine à états indifféremment du mode dans lequel elle se trouve. Si l'événement arrive alors que la machine à états est passive, elle passe alors en mode actif pour le traiter. Mais que se passe-t-il s'il est reçu pendant que la machine à états est active, en train de traiter l'événement précédent ?

L'une des principales caractéristiques des machines à états UML 2 est l'utilisation de la sémantique d'exécution des cycles *run-to-completion* dans le traitement des événements. Elle est définie comme suit dans la spécification :

> *« Event occurrences are detected, dispatched, and then processed by the state machine, one at a time. [...] The semantics of event occurrence processing is based on the* run-to-completion *assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed. »* [113]

Le traitement des événements est effectué au cours de cycles non-interruptibles[13] traitant les événements l'un après l'autre. Un cycle *run-to-completion* permet ainsi de faire passer de façon sûre, la machine à états d'une configuration stable et cohérente à une autre configuration stable et cohérente.

[13]Non-interruptible ne signifie pas ici que ce traitement monopolise le processeur mais qu'il ne peut pas être interrompu par un autre événement de la machine à états.

Ce mécanisme permet d'éviter des configurations incohérentes ou instables, comme par exemple l'activation simultanée de deux états mutuellement exclusifs, ou encore dans le cas où un état composite étant activé, la machine à états ne peut déterminer lequel de ses sous-états est actif.

3.3 PauWare.Velcro : exécutabilité et persistance à l'exécution de machines à états UML 2

La section précédente nous a permis d'introduire les modèles UML que nous utilisons pour modéliser nos composants. Issue de recherches sur le MDA (*Model-Driven Architecture*) proches d'*Executable UML*, notre approche repose sur l'exécutabilité des modèles et de leur persistance à l'exécution afin d'opérer une administration fine sur le comportement des composants.

Dans cette nouvelle section, nous présentons le MDA et *Executable UML*, puis les principes de notre environnement d'exécution de modèles, PauWare.Velcro, ainsi que son modèle de composants associé.

3.3.1 MDA et Executable UML

Model-Driven Architecture

Le MDA (*Model-Driven Architecture*) [110] et, plus généralement, le MDE[14] (*Model-Driven Engineering*) [77] préconisent l'utilisation de modèles comme entités de première classe dans le processus d'ingénierie du logiciel. L'idée sous-jacente est de ne plus seulement utiliser les langages de modélisation de façon contemplative, c'est-à-dire comme un croquis du logiciel avant son implémentation, mais de façon productive, comme un outil manipulable menant au code du logiciel.

Le MDA distingue ainsi deux types de modèles : les *Platform-Independent Models* (PIM) et les *Platform-Specific Models* (PSM). Les PIM définissent le comportement et les fonctionnalités métier du logiciel indépendamment de toutes considérations spécifiques à une plate-forme technologique particulière. Les PSM, qui sont le résultat de transformations de PIM, intègrent en plus des détails d'implémentation du logiciel spécifiques à une plate-forme donnée. La distance entre un PSM et le code doit être minimisée pour permettre la génération de code quasi-automatiquement à partir de ce dernier.

Cette approche a été proposée en 2001 par l'OMG qui, comme nous l'avons vu, est également à l'origine de la standardisation du langage de modélisation UML. C'est donc naturellement que l'OMG promeut l'usage des technologies autour d'UML pour mettre en œuvre le MDA. A l'issu d'un panel sur le MDA à OOPSLA'04 [105], Steve Cook propose de situer les différentes visions du MDA selon trois angles différents :

[14] En français IDM (Ingénierie Dirigée par les Modèles) [48].

- le premier implique l'utilisation d'UML pour construire des PIM qui sont ensuite transformés en PSM à partir desquels le code est généré,
- le second ne considère pas UML directement mais s'attache plus au MOF, le métamodèle d'UML, pour définir des langages de modélisation et de transformation,
- le troisième vise à créer une plate-forme d'exécution pour UML qui devient ainsi un langage de programmation de très haut niveau d'abstraction.

C'est dans cette troisième optique, dont *Executable UML* est le représentant majeur, que se situe PauWare.Velcro.

Executable UML

Bien que son origine provienne de l'unification de différentes méthodes orientées objet, UML n'est pas une méthode à proprement parler et on lui confère souvent l'image « d'une boîte à outils, d'une collection de concepts sans mode d'emploi » [108]. *Executable UML* [101] propose une méthodologie d'ingénierie du logciel, basée sur une utilisation d'UML permettant de définir un langage de modélisation exécutable :

> *« Executable UML [...] defines an execution semantics for a carefully selected, streamlined subset of the UML. The subset is computationally complete, so an executable UML model can be directly executed. Executable UML defines groupings of data and behavior ("classes"), the behavior over time of instances ("state charts"), and precise computational behavior ("actions")* without *prescribing implementation. »* [102]

Issus de la méthode de Shlaer et Mellor [138], les principes d'*Executable UML* ont par la suite été adaptés au MDA. Ainsi, *Executable UML* permet de construire des PIM à l'aide des diagrammes de classes, des machines à états et d'un langage d'actions conforme à la sémantique des actions dans UML (*Action Semantics*). Un compilateur de modèles permet ensuite de prendre en compte les décisions d'implémentation spécifiques à une plate-forme donnée afin de générer le code directement à partir du modèle exécutable du système. Compte tenu de l'exécutabilité des PIM, cet méthode n'utilise pas de PSM intermédiaires avant la génération du code. En fait, on peut considérer que le code est un PSM, mais un PSM qui n'est pas visualisé sous forme de diagrammes mais directement exécutable :

> *« The PSM is there alright—it's the code. Code is a weaving together of the elements of the PIM and of the required platforms, and it executes, too. Using executable models, there's no need to manipulate the PSM or to visualize it as a model. We can go straight to the code, the Mother of All PSMs. »* [102]

3.3.2 Principes de PauWare.Velcro

PauWare.Velcro fait partie de la solution de développement logiciel PauWare qui regroupe un ensemble de librairies et d'outils pour développer des applications métier en utilisant des

technologies d'ingénierie dirigée par les modèles. C'est une librairie Java ME qui permet de faciliter la construction de machines à états UML 2 exécutables sur des systèmes sans fil. Pour cela PauWare.Velcro implémente l'API (*Application Programming Interface*) qui définie le moteur d'exécution de machines à états PauWare.

Le moteur d'exécution de machines à états PauWare

Lors du passage d'un PSM au code, ou d'un PIM à un PSM dans le cas d'*Executable UML*, l'intelligibilité du modèle est souvent remise en cause par les constructions du langage de programmation. Les concepts du modèle sont enfouis dans les choix d'implémentation et le code ne reflète plus directement le modèle.

Cette distance entre le modèle et le code est particulièrement importante pour les machines à états dont l'implémentation est connue pour être difficile à réaliser :

> *« Implementing state machines is not as easy as it looks. Even with classical nonhierarchical FSMs, you must make an amazing number of design decisions and trade-offs [...] When you add state hierarchy, exit/entry actions, and transitions with guards, the design becomes anything but trivial. »* [135]

Le moteur d'exécution PauWare permet de réduire le fossé sémantique entre le modèle des machines à états et le code correspondant, de la même façon qu'un langage de programmation orienté objet intègre directement les concepts orientés objet dans les construction du langage. Cela a le double avantage de faire apparaître de façon plus explicite le modèle au niveau du code et de décomplexifier la construction des générateurs de code et autres compilateurs de modèles[15]. La figure 3.10 montre l'architecture d'une machine à états conçue avec PauWare (et par extension PauWare.Velcro).

La classe `Statechart` spécifie le type état lui-même[16] et encapsule les opérations de création d'états composites. Ainsi, un état peut être simple, *i.e.* une feuille dans la hiérarchie des états, ou composite s'il est construit à partir d'autres états grâce aux opérateurs `xor()` et `and()`. Suivant le formalisme des *Statecharts* étudié précédemment (cf. 3.2.3 p. 84), l'opérateur `xor()` traduit une décomposition XOR, c'est-à-dire par exclusion mutuelle des états, et l'opérateur `and()` traduit une décomposition AND, c'est-à-dire par concurrence. La gestion particulière de ces différents états est assurée par les mécanismes internes de la machine à états PauWare.

Le type des machines à états est spécifié par la classe `Statechart_monitor`. Dans PauWare, une machine à états est vue comme un état particulier qui est la racine de la hiérarchie des états décrite par le *Statechart* correspondant. La classe `Statechart_monitor` est donc un sous-type de `Statechart`. Cette classe ajoute par héritage les fonctionnalités nécessaires à la déclaration des transitions et à l'interprétation des événements, qui sont notamment incarnées par les méthodes `fires()` et `run_to_completion()`.

[15] PauWare fournit ainsi XMI2PauWare, un outil de transformation de machines à états UML décrites en XMI (*XML Metadata Interchange*) vers leurs formes exécutables avec la librairie PauWare.

[16] La classe est nommée `Statechart` et non pas `State` car un *Statechart* peut être récursivement un état d'un autre *Statechart*.

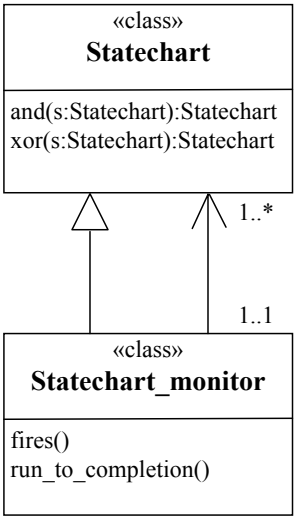


Fig. 3.10 – La machine à états PauWare

Une transition est déclarée dans une machine à états par la méthode `fires()` dont le listing 3.1 montre la signature. Cette méthode polymorphe est surchargée plusieurs fois pour traduire le caractère optionnel de certains de ses paramètres. Seuls l'état source, `from`, et l'état cible, `to`, de la transition doivent obligatoirement figurer. Les paramètres, `event`, `guard` et `action` correspondent au formalisme ECA supporté par les transitions des *Statecharts*. Les autres paramètres `guard_object`, `guard_args`, `object` et `args` permettent de désigner l'objet sur lequel l'action ou la garde doivent être appliquées et de passer la liste des arguments correspondant en paramètres.

Les événements sont interprétés par la méthode `run_to_completion()` qui, *nomen est omen*, implémente le mécanisme du *run-to-completion*. Ce mécanisme, que nous avons présenté en section 3.2.3 page 86, permet à la machine à états de traiter les événements qu'elle reçoit de

```
1  public void fires(java.lang.String event,
2                    Statechart from,
3                    Statechart to,
4                    java.lang.Object guard_object,
5                    java.lang.String guard,
6                    java.lang.Object[] guard_args,
7                    java.lang.Object object,
8                    java.lang.String action,
9                    java.lang.Object[] args)
10     throws Statechart_transition_based_exception
```

List. 3.1 – Signature de la méthode `fires`

façon sûre, en assurant la cohérence de l'état global de la machine à états d'un cycle *run-to-completion* à l'autre.

Cette méthode prend donc en paramètre le nom de l'événement à traiter et calcule en conséquence le nouvel état global courant. Afin de suivre le principe du *run-to-completion*, il faut que ce traitement ne puisse pas être interrompu. Pour cela, la méthode est déclarée *synchronized*, ce qui pose un verrou Java sur la machine à états. La machine à états peut ainsi traiter sûrement l'événement en cours, tout en recevant les événements suivants, automatiquement mis en file d'attente par le mécanisme de *multithreading* de Java.

Certains défauts de conception peuvent amener la méthode `run_to_completion()` à détecter un état global incohérent à la fin d'un cycle. Une exception de type `Statechart_exception` est alors levée. Lors d'une simulation de machines à états, le traitement de cette exception permet d'analyser les causes de cette erreur.

Exemple descriptif de l'utilisation de l'API

L'API de PauWare, et de PauWare.Velcro, est décrite dans un manuel d'utilisation complet de plus de 70 pages [11]. Nous présentons ici les principales fonctionnalités de l'API à travers un exemple simple et renvoyons le lecteur au manuel d'utilisation pour de plus amples détails et d'autres exemples plus complexes.

L'exemple que nous avons choisi pour décrire l'utilisation de PauWare est celui d'une lampe `Light`, dont le comportement est spécifié par la machine à états de la figure 3.11. L'implémentation correspondante de cette machine à états avec PauWare est présentée dans le listing 3.2.

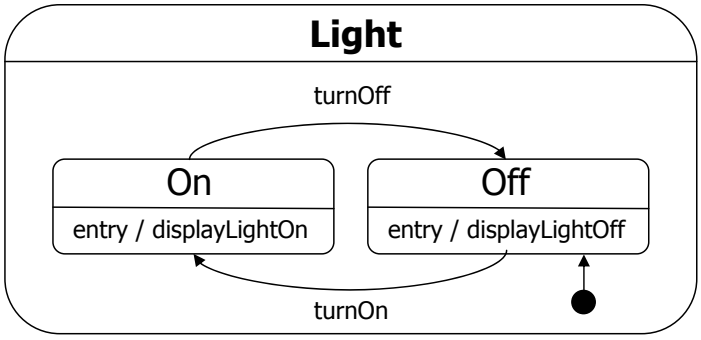


FIG. 3.11 – Machine à états du comportement d'une lampe `Light`

La lampe a deux états possibles. Elle peut être soit dans l'état *On* (allumée), soit dans l'état *Off* (éteinte), et son état initial est l'état *Off*. Cela se traduit par l'instanciation de deux objets `Statechart` aux lignes 2 et 5, et par l'appel de la méthode `inputState()` sur l'état `off`.

Ces deux états possèdent une action d'entrée, respectivement `displayLightOn` et `displayLightOff`, qui provoque l'affichage adéquat sur l'interface graphique de lampe. Chaque action

```
1  /* UML statecharts */
2  Statechart on = new Statechart("On");
3  on.entryAction(lightGUI,"displayLightOn");
4
5  Statechart off = new Statechart("Off").inputState();
6  on.entryAction(lightGUI,"displayLightOff");
7
8  /* PauWare engine */
9  final Statechart_monitor light =
10     new Statechart_monitor(on.xor(off),"Light");
11
12 /* UML transitions */
13 Light.fires("turnOn",off,on);
14 Light.fires("turnOff",on,off);
```

LIST. 3.2 – Implémentation de la machine à états `Light`

est ajoutée à l'état correspondant grâce à la méthode `entryAction()`. Comme pour les actions portées par les transitions qui sont définies par la méthode `fires()`, les actions portées par les états, *i.e.* les actions *entry & exit* et les activités *do*, permettent de spécifier sur quel objet doit être opérée l'action. Ici ces deux actions sont appliquées sur l'interface graphique de la lampe `lightGUI`[17].

Aux lignes 9 et 11, la machine à états `light` est instanciée. Elle est décrite comme un état composite des états *On* et *Off* qui sont composés par exclusion mutuelle (`on.xor(off)`). Une fois que la hiérarchie des états est décrite, les transitions sont ensuite spécifiées. Les lignes 13 et 14 déclarent respectivement une transition de l'état *Off* vers l'état *On* déclenchée par l'événement `turnOn`, et une transition de l'état *On* vers l'état *Off* déclenchée par l'événement `turnOff`. La machine à états de la figure 3.11 est alors complètement décrite avec l'API PauWare et son exécution peut commencer.

L'exécution des machines à états est dirigée par les événements qu'elles reçoivent. L'exemple du listing 3.3 montre une utilisation concurrente de la machine à états `light`. Chacun des deux événements de `light` est généré via la méthode `run_to_completion()` par un *thread* Java qui envoie un nouvel événement toutes les 2 secondes dans le cas de `turnOn` et toutes les 3 secondes pour `turnOff`. La première instance de l'événement `turnOn` apparaît donc au temps t2[18], ce qui fait transiter la machine à états de l'état *Off* à l'état *On*. La première instance de l'événement `turnOff` apparaît au temps t3, ce qui positionne à nouveau `light` dans l'état *Off*, et ainsi de suite jusqu'au temps t6 où la machine à états reçoit en concurrence une instance de `turnOn` et une instance de `turnOff`. Ces deux événements sont automatiquement mis en file d'attente par le service de *multithreading* Java et traités selon le principe du *run-to-completion*. L'ordre des

[17] L'implémentation de la partie graphique est omise dans le code.
[18] tn = t0 + n secondes.

```
/* Event-driven execution */
Thread t1 = new Thread() {
    public void run() {
        try {
            while(true) {
                this.sleep(2000);
                /* UML event: send turnOn */
                light.run_to_completion("turnOn");
            }
        } catch (Exception e) {/*...*/}
    }
};
Thread t2 = new Thread() {
    public void run() {
        try {
            while(true) {
                this.sleep(3000);
                /* UML event: send turnOff */
                light.run_to_completion("turnOff");
            }
        } catch (Exception e) {/*...*/}
    }
};
t1.start(); t2.start();
```

List. 3.3 – Simulation de l'exécution de la machine à états `Light`

événements dans la file n'étant pas déterministe, il a deux traces d'exécution possibles comme le montrent les chronogrammes des figures 3.12 et 3.13.

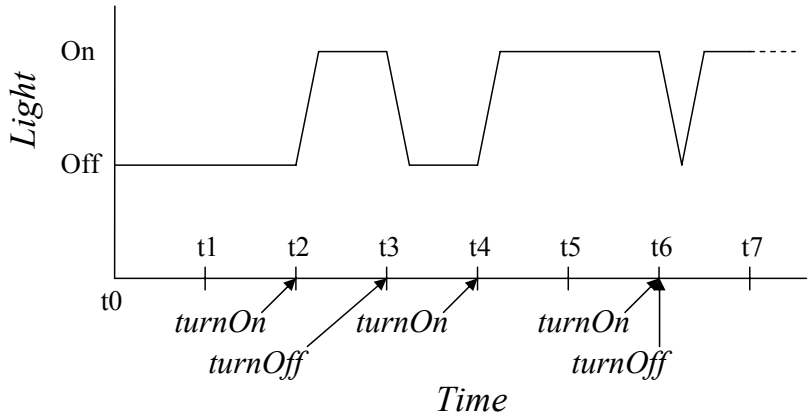


FIG. 3.12 – Trace d'exécution avec traitement de l'événement `turnOff` avant l'événement `turnOn` au temps t6 (l'événement `turnOff` est pris en compte)

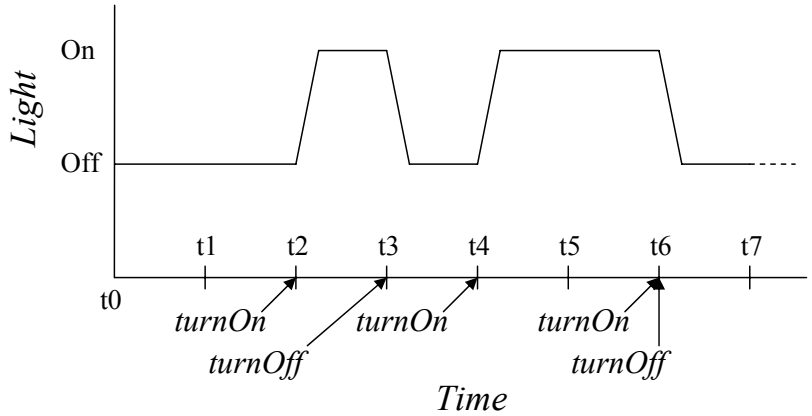


FIG. 3.13 – Trace d'exécution avec traitement de l'événement `turnOff` après l'événement `turnOn` au temps t6 (l'événement `turnOff` n'a pas d'effet)

L'état courant étant l'état *On*, si l'événement `turnOff` est traité en premier alors la machine à états va transiter vers l'état *Off* puis l'événement `turnOn` sera traité subséquemment ce qui provoquera le retour à l'état *On* (cf. figure 3.12). Au contraire, si l'événement `turnOn` est traité en premier, il ne provoquera pas de changement d'état car l'état courant est *On* et aucune transition ne peut être déclenchée par cet événement à partir de cet état[19]. L'événement `turnOff` sera alors traité et activera le passage à l'état *Off* (cf. figure 3.13).

Cet exemple montre la robustesse de la machine à états PauWare face à l'indéterminisme de

[19]Cela résulte du principe de communication *broadcast* des *Statecharts* dans lequel un événement peut être envoyé sans savoir dans quel état se trouve le receveur.

l'arrivée des événements ce qui est une propriété particulièrement intéressante pour des systèmes dont l'environnement dynamique n'est pas toujours prévisible. La machine à états garantit que le comportement spécifié sera respecté. Dans notre exemple, un appel à l'opération `displayLightOn` succédera toujours un appel à `displayLightOff` et inversement. Même si plusieurs événements `turnOn` successifs, ou `turnOff` successifs, sont envoyés à `light`, cette alternance sera respectée.

3.3.3 Principes du modèle de composants PauWare

Le principe fondamental du modèle de composants PauWare est d'incorporer à chaque composant une machine à états PauWare décrivant son comportement. Ce principe peut d'ailleurs être appliqué à différents modèles de composants déjà existants, comme par exemple dans [10] où la machine à états PauWare est intégrée dans des composants EJB. Nous avons ainsi choisi de concevoir une structure simple et neutre pour notre modèle, à base d'interfaces et de classes comme cela peut se faire dans les modèles de composants POJO, de façon à pouvoir transposer facilement notre travail sur le comportement vers un autre modèle de composants.

La figure 3.14 présente la structure canonique d'un composant PauWare. Cette structure est exprimée avec des diagrammes de composants et de structures composites UML 2.

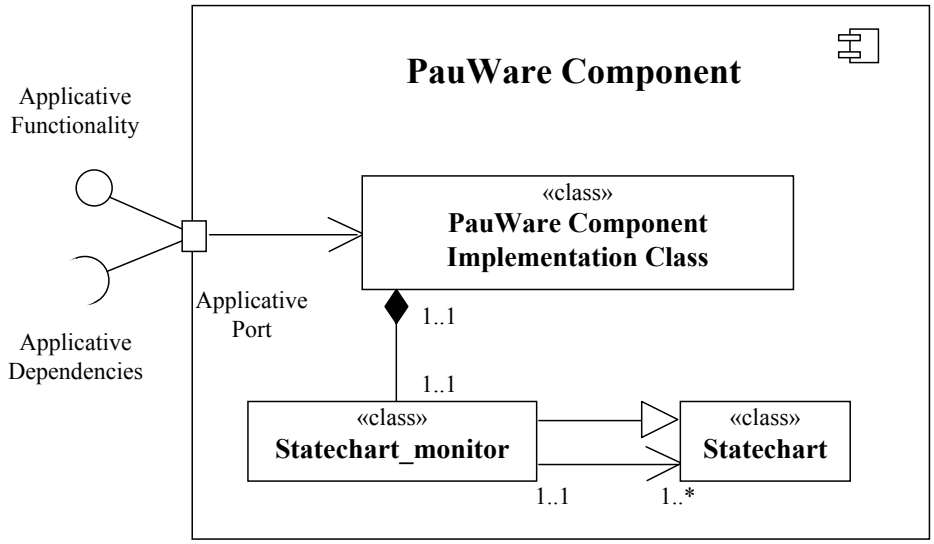


FIG. 3.14 – Un composant canonique PauWare intègre une machine à états pour réaliser son comportement

Comme le définit son métamodèle, UML 2 permet d'associer un modèle de comportement à un classifieur et par conséquent à un composant. Par contre, la spécification ne détermine pas la façon de le faire et ne limite pas non plus son utilisation ni son but [137]. Pour définir le modèle de composants PauWare, notre travail doit préciser les liens entre la structure et le

comportement de nos composants, c'est-à-dire les liens entre la machine à états PauWare d'une part, et les interfaces et la classe d'implémentation d'un composant d'autre part.

Les figures 3.15 et 3.16 spécifient en UML un exemple de composant PauWare, nommé *PauWare Component*, respectivement du point de vue de sa structure et du point de vue de son comportement. Le comportement représenté par le *Statechart* de la figure 3.16 est exécuté par une machine à états intégrée au composant, qui correspond au *Statechart_monitor* et aux différents états (`Statechart`) inclus comme champs privés de la classe d'implémentation du composant de la figure 3.15.

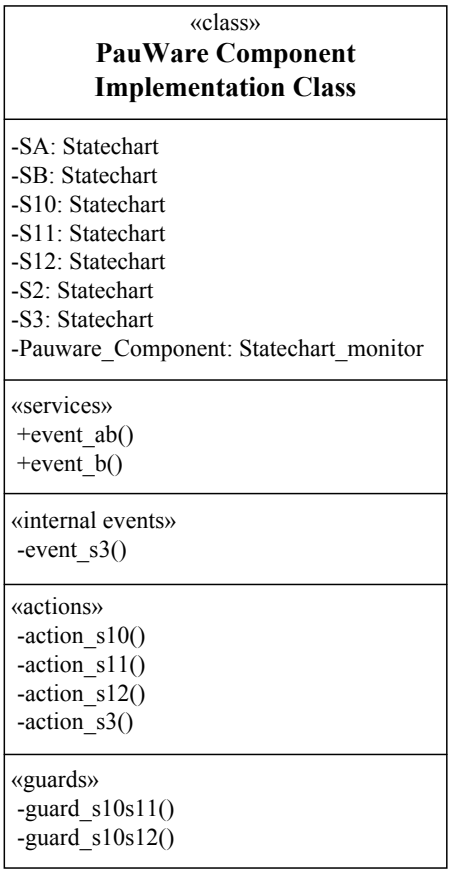


FIG. 3.15 – Implémentation du composant PauWare

Les services du composant (`event_ab()`, `event_b()`, `event_s3()`) correspondent aux événements du même nom dans le diagramme du comportement. Le code de ces services encapsule dans des méthodes de la classe d'implémentation des appels à la méthode `run_to_completion()` de la machine à états, comme le montre le listing 3.4.

Les services qui figurent dans le compartiment « *services* » sont accessibles de l'extérieur. Ils font partie de l'interface de services fournis du composant. Certains services peuvent n'être utilisés seulement qu'en interne, comme `event_s3` dans le compartiment « *internal events* ». Ils correspondent alors à des événements envoyés par la machine à états sur elle-même. Ils

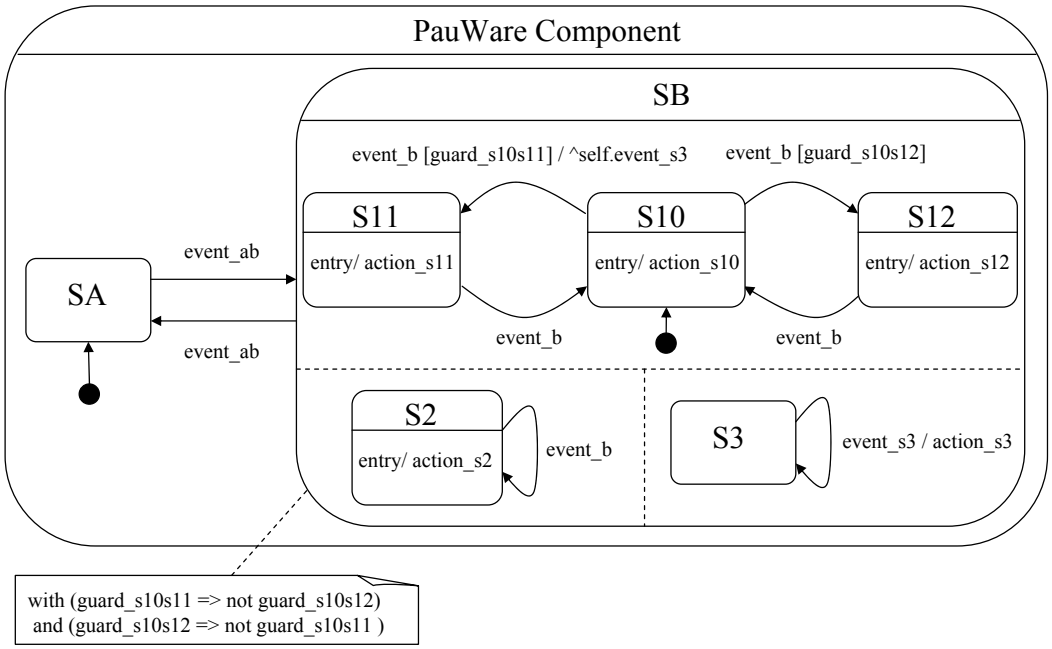


FIG. 3.16 – Comportement du composant `PauWare Component`

```
public void event_ab() {
    PauWare_Component.run_to_completion("event_ab");
}

public void event_b() {
    PauWare_Component.run_to_completion("event_b");
}

public void event_s3() {
    PauWare_Component.run_to_completion("event_s3");
}
```

LIST. 3.4 – Implémentation des services du *PauWare Component* comme événements de sa machine à états

```
public boolean guard_s10s11() {
    if(this.i <= this.I_MAXIMUM_LIMIT) {
        return true;
    } else {
        return false;
    }
}

public boolean guard_s10s12() {
    if((this.J_MINIMUM_LIMIT <= this.j)
       && (this.j <= this.J_MAXIMUM_LIMIT)) {
        return true;
    } else {
        return false;
    }
}
```

LIST. 3.5 – Implémentation des gardes du *PauWare Component*

n'apparaissent donc pas dans l'interface de services fournis du composant, mais sont réservés à un usage interne.

Les méthodes qui encapsulent des algorithmes contenus par le composant apparaissent dans le compartiment « *actions* » de la classe d'implémentation. Elles correspondent à des actions du *Statechart* du composant qui sont orchestrées par la machine à états suivant le comportement spécifié.

L'accès à certaines données internes au composant dans le calcul de certaines gardes de la machine à états, impose l'ajout de méthodes supplémentaires. C'est le cas des gardes `guard_s10s11` et `guard_s10s12` de notre exemple, dont le code est présenté dans le listing 3.5. Cela correspond à une instrumentation interne du composant comme nous le verrons dans le chapitre suivant.

Lorsqu'un composant est connecté à un autre composant par une interface de services requis, les services requis du composant correspondent à des signaux du *Statechart* du composant. Ces signaux sont envoyés au composant connecté qui les interprète comme des événements de sa machine à états. L'exemple de la figure 3.17, qui connecte un composant `Controller` à un composant `Gate`, illustre cette situation. Le signal ^`gate.switch` de la machine à états du composant `Controller` correspond à l'événement `switch` de la machine à états du composant `Gate`.

Ainsi les fondements du modèle de composants PauWare reposent sur la réification du comportement des composants en *Statecharts* exécutables et accessibles ce qui va nous permettre d'administrer de manière fine le comportement de ces composants.

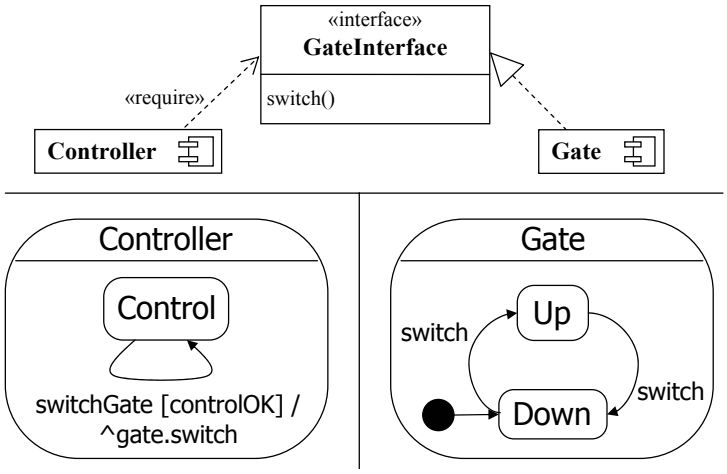


FIG. 3.17 – Relations et comportements des composants `Controller` et `Gate`

Le modèle de composant PauWare permet également d'envisager des mécanismes de composition prenant en compte le comportement des composants. Par exemple dans [129], nous avons proposé une technique de composition verticale pour les composants PauWare, basée sur la composition des *Statecharts* des composants. Le principe de cette composition repose sur la relation récursive entre état et machine à états : au niveau d'un composant composite, les machines à états de ses sous-composants sont considérées comme des états qui composent sa machine à états. Dans notre approche, le comportement global d'un composant composite est le résultat de la composition orthogonale (AND) du comportement des sous-composants et du comportement propre du composite. L'exemple de la figure 3.18 décrit la machine à états du composite `Composite Component` et de ses deux sous-composants `First Subcomponent` et `Second Subcomponent`.

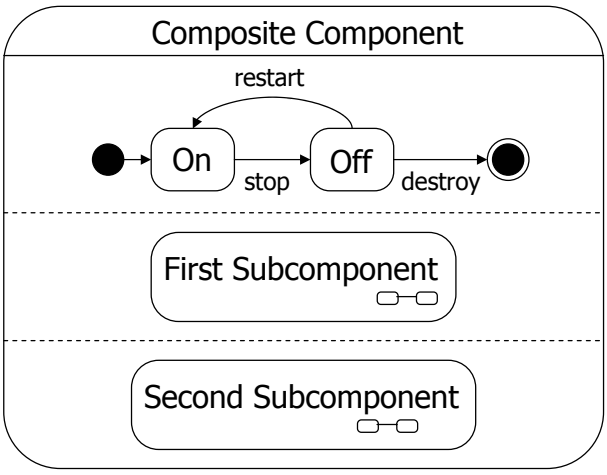


FIG. 3.18 – Comportement d'un composant composite dans PauWare

3.4 Synthèse

Nous avons présenté dans ce chapitre notre approche de conception de composants logiciels permettant de faire persister le modèle de leur comportement, en vue de leur administration à l'exécution. Cette approche s'appuie sur les diagrammes de composants et de structures composites d'UML 2 pour représenter la structure de nos composants et sur les diagrammes de machines à états pour représenter leur comportement.

Grâce à la librairie d'exécution de modèles PauWare, les composants intègrent une machine à états exécutable qui permet de réifier le comportement des composants à l'exécution. Ainsi nous avons réduit le problème de l'administration du comportement des composants logiciels en général, à l'administration de la machine à états PauWare en particulier, puisqu'elle permet d'exécuter n'importe quel comportement modélisé à partir de *Statecharts*.

Le but du chapitre suivant consistera donc à définir l'architecture interne nécessaire pour réaliser un composant administrable, tel qu'illustré par la figure 3.19, ainsi que l'architecture globale d'administration dans un contexte sans fil.

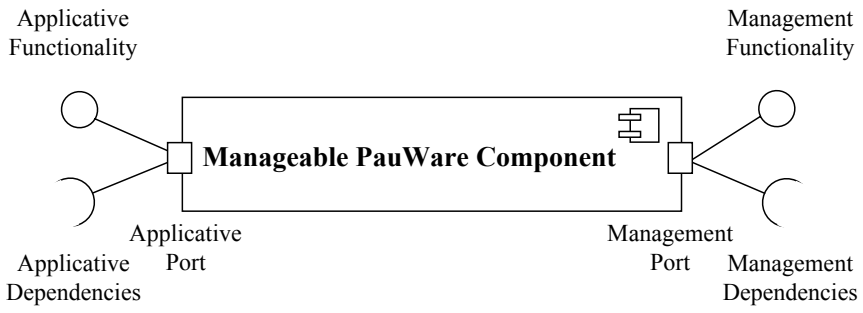


Fig. 3.19 – Un composant PauWare administrable

Après avoir traité la partie applicative dans ce chapitre, nous verrons comme créer un port d'administration pour nos composants dans la suite de ce manuscrit.

Chapitre 4

Une architecture d'administration de composants logiciels sans fil

Sommaire

4.1 Architecture d'administration 102
4.1.1 Les architectures possibles 102
4.1.2 L'architecture retenue 105
4.2 Administration interne 106
4.2.1 Gestionnaire interne et composant métier 107
4.2.2 Capteurs et effecteurs internes 108
4.3 Administration externe 111
4.3.1 Réplication des machines à états 111
4.3.2 Capteurs et effecteurs externes 114
4.3.3 Contrôle à distance 118
4.4 Exemple d'une politique pour la gestion de compositions 119
4.4.1 Composition comportementale avec les *Statecharts* et la relation Tout-Partie 119
4.4.2 Une politique d'administration pour assurer la composition rigoureuse du comportement 121
4.5 Synthèse 124

Ce chapitre présente notre architecture d'administration pour les composants logiciels déployés sur des systèmes sans fil. Notre objectif étant de permettre une gestion fine du comportement des composants, nous prenons en compte les besoins de l'administration dès la conception, de sorte que le comportement des composants soit accessible à l'exécution. Notre approche s'appuie sur une technique d'exécution de modèles comme nous l'avons étudié au chapitre précédent. Le comportement de chaque composant est d'abord spécifié avec des *Statecharts* puis matérialisé à l'exécution par une machine à états construite avec la librairie PauWare.

Nous verrons dans ce chapitre les choix architecturaux que nous avons fait pour administrer les composants sans fil à travers leur machine à états. Nous étudierons en particulier les mécanismes de gestion interne de ces composants, puis leur administration distante via la réplication

de leur comportement dans le système d'administration. Nous conclurons ce chapitre par un exemple de politique administration appliquée sur les composants sans fil par l'intermédiaire de leurs répliques.

4.1 Architecture d'administration

Lors de la présentation générale de l'approche au chapitre 3 (cf. section 3.1 p. 74), nous avons ébauché notre architecture d'administration. Dans cette section, nous revenons sur les différentes architectures possibles, puis nous détaillons l'architecture que nous avons retenue.

4.1.1 Les architectures possibles

Globalement, notre solution pour l'administration des composants logiciels sans fil se présente comme un système distribué, en environnement sans fil, composé de deux nœuds : celui du système sans fil et celui du système d'administration. Pour administrer le comportement d'un composant dans notre approche, le système d'administration doit accéder à la machine à états qui exécute le comportement du composant. Le système sans fil et le système d'administration ont donc tous les deux besoin des informations de la machine à états, respectivement pour le fonctionnement métier de l'application et pour son administration.

La question du choix du nœud de déploiement pour la machine à états se pose alors : directement dans le système d'administration ou embarqué dans le système sans fil ?

Logique du comportement déportée à distance

Dans sa thèse sur l'administration, Buzato fait valoir une architecture des systèmes distribués séparant la logique de contrôle, *i.e.* le comportement, et la logique applicative sous forme d'actions :

> *« From the point of view of the management system, a distributed program is seen as the superposition of two reactive programs : an application program and a management program. [...] At runtime, the objects of a control program constantly monitor the objects of an application program and allow them to adapt to changes occurring in their environment [...]. »* [24]

Le comportement de l'application est encapsulée dans le *Control Program* qui fait partie du système d'administration. Le *Control Program* coordonne ainsi les actions de l'*Application Program* selon le comportement spécifié et les règles d'administration. La figure 4.1 illustre cette architecture.

Appliquée à notre approche, une architecture similaire revient à placer la machine à états PauWare d'un composant à administrer sous le contrôle d'un gestionnaire directement dans le

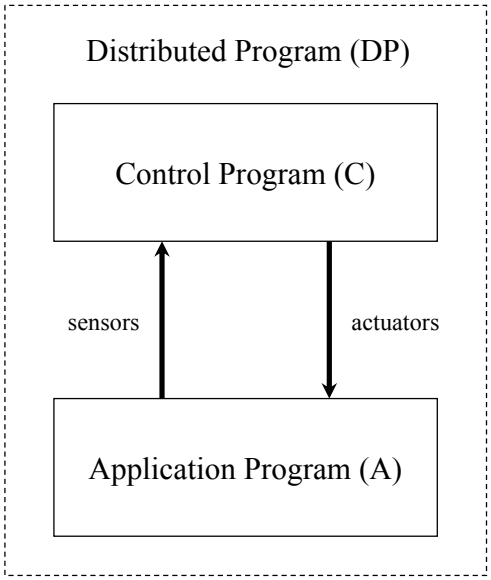


FIG. 4.1 – Un programme distribué vu comme un programme réactif (extrait de [24])

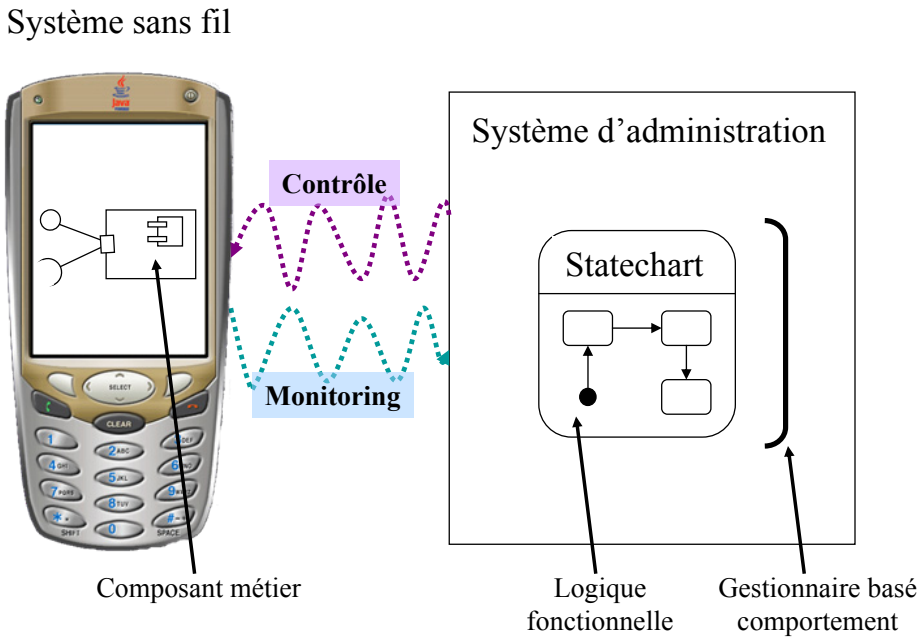


FIG. 4.2 – Architecture avec logique du comportement distante

système d'administration. La machine à états est alors externe au composant et le composant ne comporte que des fonctionnalités métier. Chaque appel de service sur le composant doit alors être transféré au gestionnaire qui, à partir de la machine à états, ordonne au composant les actions à exécuter en fonction de l'événement reçu. L'architecture correspondante est représentée en figure 4.2.

Dans notre contexte d'administration des composants logiciels sans fil, cette architecture présente plusieurs inconvénients. Premièrement, la réactivité du composant à un appel de service est temporisée par la communication synchrone nécessaire entre le composant et son gestionnaire. Le composant doit d'abord transférer l'événement qu'il reçoit et attendre la réponse de son gestionnaire pour pouvoir exécuter l'action correspondante. Cela est d'autant plus dommageable que les communications sans fil permettent un débit moindre que dans les réseaux filaires.

Deuxièmement, la séparation physique entre le composant et sa logique fonctionnelle rend le composant totalement dépendant du système d'administration. Cela va à l'encontre de notre vision du composant logiciel administrable qui doit posséder une certaine autonomie et une cohérence fonctionnelle vis-à-vis du système d'administration. Les dépendances avec un système d'administration doivent rester optionnelles et ne pas nuire à l'exécution de l'application.

Logique du comportement embarquée

A l'inverse de l'architecture précédente, la machine à états PauWare peut être embarquée dans le système sans fil grâce à son extension PauWare.Velcro. Ainsi, il est possible de reproduire le modèle de composant PauWare tel que nous l'avons présenté au chapitre précédent (cf. 3.3.3 p. 95). La problématique est alors réduite à l'administration du modèle de composants PauWare.

A l'instar d'OSGi, il est envisageable de déployer une infrastructure d'administration directement dans le système sans fil. Le système d'administration distant se résume alors à une simple console d'administration car tous les mécanismes de gestion sont intégrés dans le système sans fil. La figure 4.3 schématise cette architecture pour notre approche.

Cependant la machine à états des composants PauWare n'est pas administrable en l'état. Il est malgré tout possible d'envisager avec cette architecture une observation externe des composants via leurs interfaces fonctionnelles. Du fait de l'aspect « boîte noire » des composants, le comportement n'est alors saisi que de façon externe et peu précise, et l'administration est limitée l'activité de surveillance : on se heurte ici au même problème que les approches basées sur le comportement que nous avons étudiées au chapitre 2 (cf. 2.3.3 p. 58).

Ainsi notre objectif de permettre le contrôle du comportement n'est pas possible avec une telle hypothèse. Bien que l'encapsulation apportée par les composants « boîte noire » soit propice à un développement fiable du logiciel [120], elle est par contre préjudiciable pour l'administration [157].

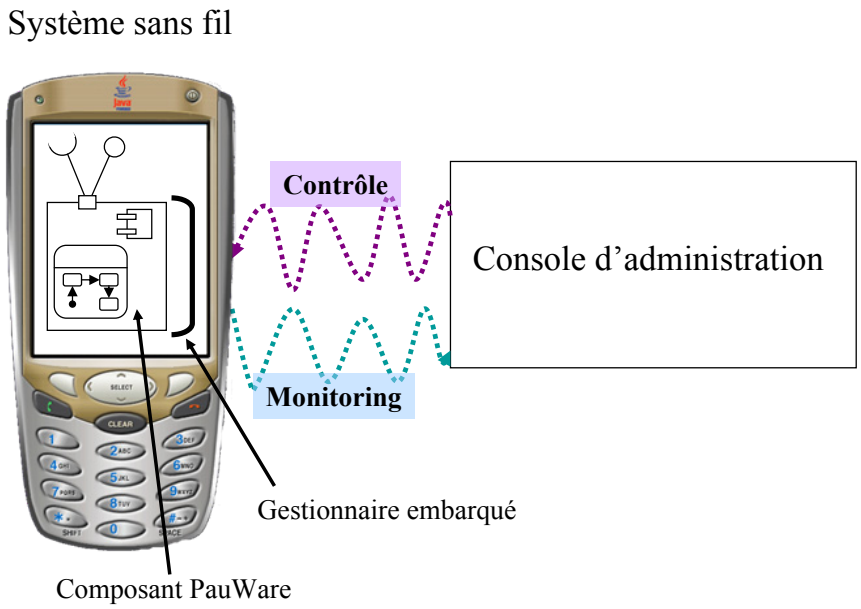


FIG. 4.3 – Architecture avec logique du comportement embarquée

4.1.2 L'architecture retenue

Pour remédier au problème de l'accès au comportement des composants, autrement dit le problème de l'accès à leur machine à états, nous proposons d'aménager proprement l'encapsulation des composants. Par « proprement » nous entendons que seul le système d'administration doit pouvoir contrôler la machine à états, sans pour autant avoir accès à tous les détails internes du composant.

L'architecture que nous avons retenue s'appuie sur l'étude des deux architectures que nous venons de voir. A l'instar de la deuxième approche, nous proposons une administration locale au composant sans fil, mais une administration partielle. Notre architecture repose sur l'intégration dans le composant administré d'un gestionnaire interne. Ce gestionnaire interne est à la fois responsable du contrôle de la machine à états du composant et de la communication avec le système d'administration. Un composant muni d'un tel gestionnaire est nommé composant WMX (*Wireless Management eXtensions*), du nom de la technologie que nous avons développée et dont nous présenterons les particularités d'implémentation au chapitre 5.

Les décisions d'administration s'effectue à distance au sein du système d'administration comme dans la première architecture présentée dans la section précédente. De cette façon, le système sans fil n'a pas à supporter la surcharge d'une infrastructure complète d'administration comme dans la deuxième architecture. La machine à états du composant est répliquée dans le système d'administration pour permettre la gestion à distance via un gestionnaire externe. Ainsi, la logique du comportement est embarquée dans le système sans fil mais elle est également accessible à distance et virtuellement présente dans le système d'administration. La figure 4.4 résume cette architecture.

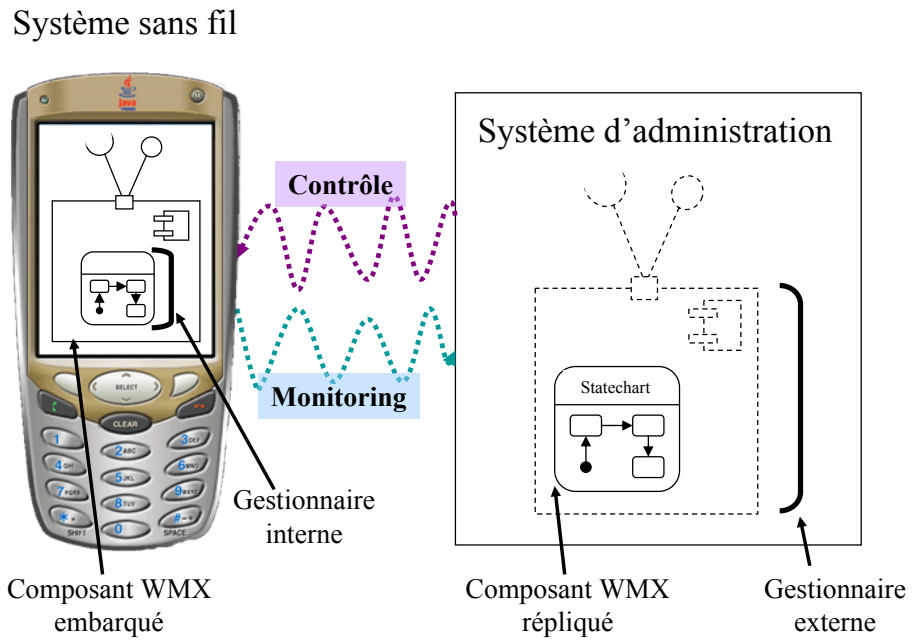


FIG. 4.4 – L'architecture retenue

La réplication de la machine à états des composants que nous avons mise en place n'est pas seulement statique. Pour permettre l'administration du comportement à l'exécution, la co-évolution de la machine originale et de sa réplique doit être assurée. Cela impose une synchronisation entre la machine à états déployée dans le système sans fil et celle déployée dans le système d'administration. Nous examinerons les détails de ce mécanisme dans la section 4.3.1.

Cette architecture fait également apparaître deux niveaux d'administration :

- une administration interne par l'intermédiaire du gestionnaire interne qui contrôle la machine à états incorporée dans le composant administré, et
- une administration externe par l'intermédiaire du gestionnaire externe qui manipule la machine à états répliquée dans le système d'administration.

Cela offre une transparence pour le système d'administration qui pourra administrer différents types de ressources en concevant différents adaptateurs pour relier l'administration interne et l'administration externe. Ces deux niveaux d'administration sont présentés dans les sections 4.2 et 4.3.

4.2 Administration interne

Dans cette section, nous présentons les composants WMX et leur administration locale au sein des systèmes sans fil. Ces composants comprennent un composant métier qui encapsule leurs fonctionnalités applicatives et un gestionnaire interne qui contrôle leur logique comportementale.

L'administration interne s'effectue grâce à des capteurs et des effecteurs qui permettent les relations au sein des composants entre leur composant métier et leur gestionnaire interne.

4.2.1 Gestionnaire interne et composant métier

Les composants WMX sont une extension du modèle de composants PauWare pour l'administration. Leur développement suit les mêmes principes : le comportement des composants est modélisé avec des machines à états UML 2 qui sont incorporées dans les composants pour être ensuite directement exécutées. La librairie PauWare.Velcro, qui étend la librairie PauWare au domaine du sans fil[20], est utilisée dans les composants WMX pour construire et exécuter leur machine à états.

Pour permettre l'administration des composants via leur machine à états, la structure interne des composants WMX a été revue par rapport au modèle PauWare. Un composant WMX se compose ainsi de deux sous-composants :

- son composant métier, et
- son gestionnaire interne.

Cette structure est décrite par la figure 4.5.

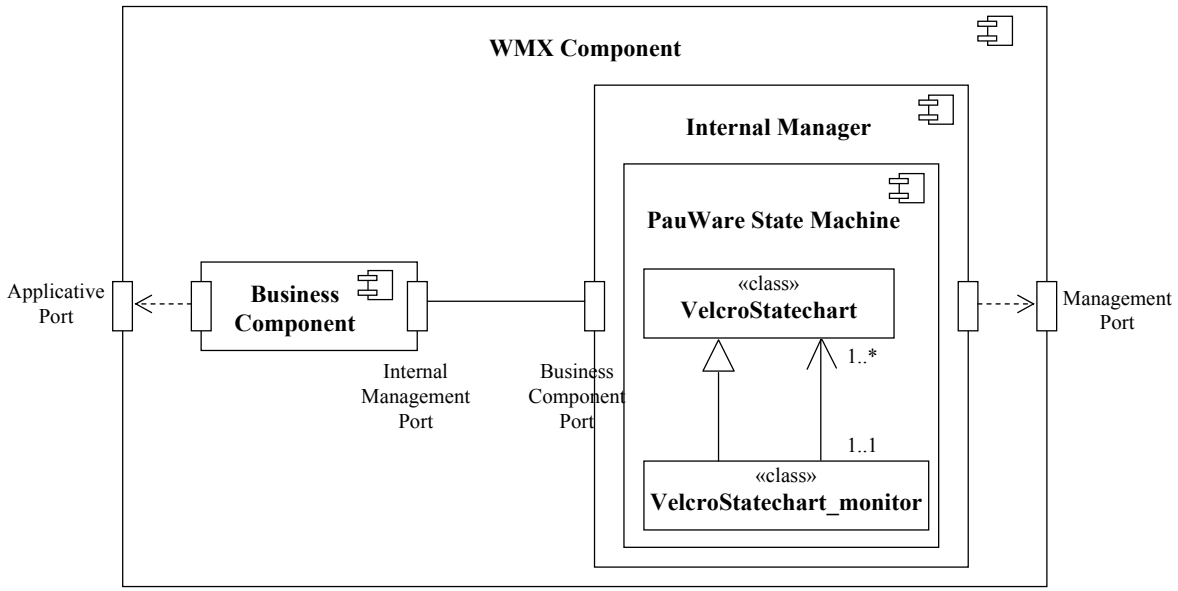


Fig. 4.5 – Structure interne d'un composant WMX

Le composant métier encapsule les données métier et les traitements que le composant peut exécuter. La logique de contrôle qui définit les enchaînements entre les traitements du composant métier est contenue par le gestionnaire interne qui encapsule la machine à états PauWare définissant le comportement du composant. Cette dichotomie entre données et traitements, d'une

[20] cf. chapitre 5 pour des détails d'implémentation techniques.

part, et contrôle, d'autre part, est inspirée de l'architecture d'administration de Buzato que nous avons présentée en section 4.1.1. C'est également ce que l'on retrouve dans [88] où la logique de contrôle des composants est externalisée dans des connecteurs.

Dans les composants WMX, la machine à états PauWare, qui matérialise le comportement des composants, n'est pas externalisée mais elle est clairement séparée dans leur structure. Cette séparation permet au gestionnaire interne de contrôler l'utilisation de la machine à états tant au niveau applicatif pour la logique de contrôle qu'au niveau de l'administration pour observer et manipuler cette logique. Le gestionnaire interne donne ainsi l'accès à la machine à états, au composant métier via le port `Business Component Port` et au système d'administration via le port `Management Port`.

Le gestionnaire interne permet également d'élever le niveau d'abstraction des informations d'administration qui sont communiquées au système d'administration par l'observation et le contrôle des données et des traitements du composant métier qui constituent le comportement bas niveau du composant. Cette administration interne s'effectue à travers les capteurs et les effecteurs qui définissent le port `Business Component Port` du composant métier et dont les détails sont précisés dans la section suivante.

4.2.2 Capteurs et effecteurs internes

Nous avons vu au chapitre 2 que, dans les solutions classiques d'administration, la ressource administrée et le système d'administration interagissent via des capteurs et des activateurs[21]. Les capteurs sont utilisés par les gestionnaires pour sonder les données de la ressource et les activateurs sont utilisés pour effectuer des actions sur la ressource.

Comme les composants logiciels communiquent à travers leurs interfaces, il est naturel de spécifier de tels capteurs et activateurs en tant qu'interfaces des composants à administrer. C'est le choix que l'on retrouve par exemple dans l'*open implementation* de Kiczales [80, 81] qui définit une interface supplémentaire pour ajuster les décisions d'implémentation stratégique, ou encore dans [84] qui définit également une interface spécifique, appelée *Diagnostic and Management interface*, qui permet un accès sélectif aux constructions internes des composants pour leur administration.

Nous avons également fait le choix d'utiliser des interfaces pour les capteurs et les effecteurs de nos composants. La figure 4.6 représente l'architecture détaillée de la version WMX du composant `PauWare Component` que nous avons présenté au chapitre 3. Pour faciliter la lecture, nous reproduisons ici en figure 4.7 le modèle du comportement du `WMX Component`.

Cet exemple illustre les relations entre le gestionnaire interne et le composant métier qu'il administre. Nous distinguons deux sortes de données devant être capturées par le gestionnaire : les requêtes de services et les états de bas niveau. Nous appelons « états de bas niveau » les valeurs des attributs du composant métier. Ces états, qui sont traditionnellement observés dans

[21] Egalement appelés effecteurs.

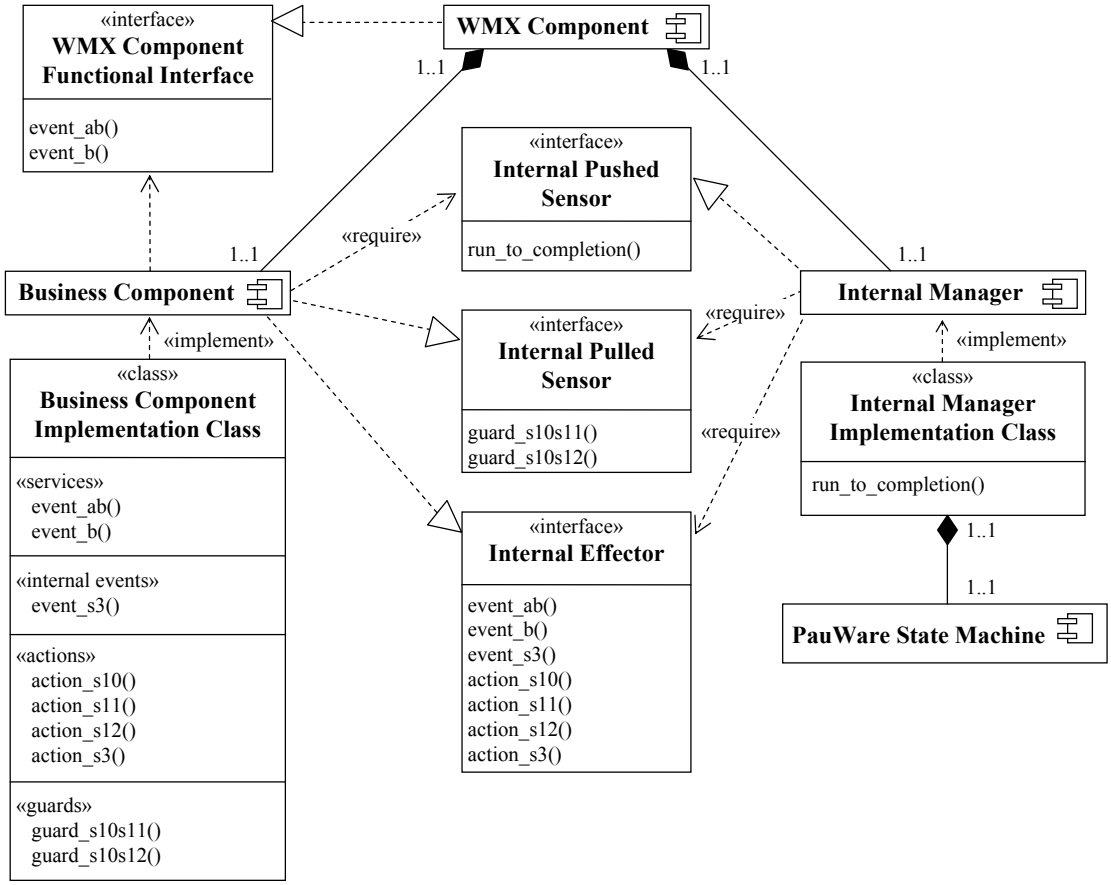


FIG. 4.6 – Architecture détaillé du composant `WMX Component`

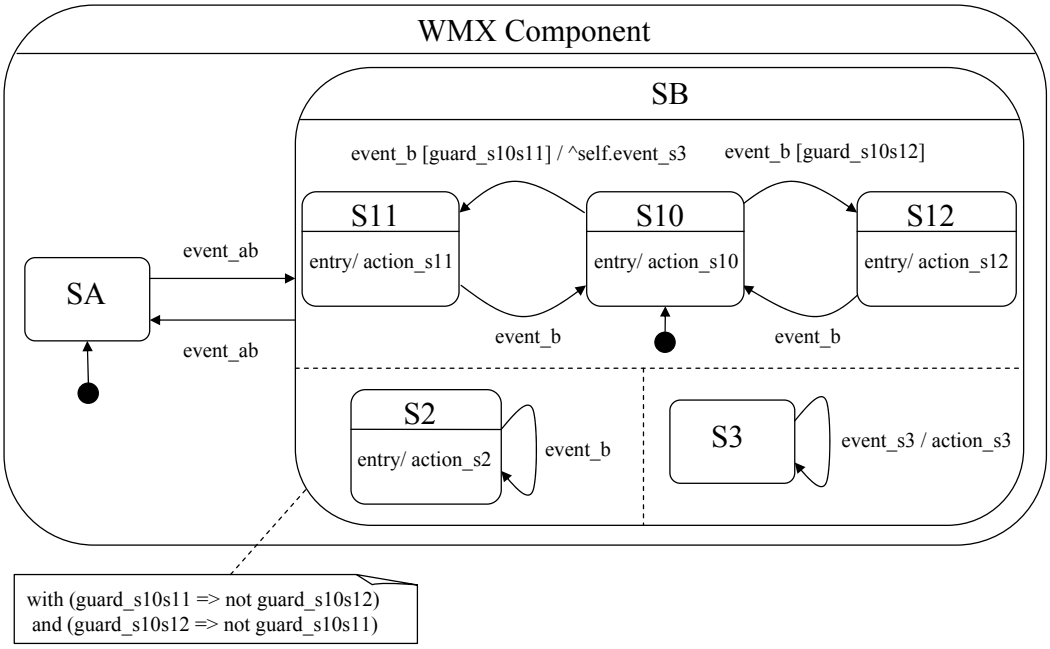


FIG. 4.7 – Comportement du composant `WMX Component`

les différentes approches d'administration, sont ici transformés en informations plus abstraites par l'évaluation des gardes du *Statechart* dans lesquelles ils interviennent.

Pour réaliser l'observation des requêtes de services et des états de bas niveau, nous utilisons les deux modèles utilisés classiquement dans les systèmes d'administration pour collecter des donnéees [92] :

- le modèle *push*, et
- le modèle *pull*.

Le modèle *pull* est basé sur le paradigme requête/réponse. Dans ce modèle, le gestionnaire envoie une requête de données à la ressource administrée, puis cette dernière répond en envoyant les données demandées. Dans un composant WMX, l'interface de services fournis `Internal Pulled Sensor`, dont dispose son composant métier, est un capteur basé sur ce modèle. Ce capteur est utilisé pour évaluer les gardes du *Statechart*.

Le modèle *push* est basé sur le paradigme *publish/subscribe*. Dans ce modèle, le gestionnaire indique à la ressource les données qu'il veut observer. Lorsqu'une donnée change de valeur, la ressource doit alors l'envoyer au gestionnaire. Un capteur basé sur ce modèle est parfaitement adapté pour notifier le gestionnaire interne de l'arrivée d'une nouvelle requête de service. L'interface de service `Internal Pushed Sensor` est utilisée dans ce but.

Enfin, une troisième interface joue le rôle d'effecteur pour permettre au gestionnaire d'activer l'exécution d'actions sur le composant métier ou pour générer des événements sur ce dernier : c'est l'interface `Internal Effector`.

4.3 Administration externe

Nous avons vu que le gestionnaire interne d'un composant WMX est responsable du contrôle de la machine à états intégrée au composant. Pour permettre la gestion du composant depuis le système d'administration distant, la machine à états du composant est répliquée dans un gestionnaire externe. La synchronisation entre les deux machines à états est maintenue grâce au transfert des informations d'administration via les capteurs et effecteurs externes du gestionnaire interne.

4.3.1 Réplication des machines à états

La réplication des machines à états agit comme un cache au niveau du système d'administration. Plutôt que d'accéder directement à l'état du composant WMX sur le système sans fil, l'activité d'administration est exercée à partir d'une copie de cet état. La machine à états répliquée est mise à jour par le gestionnaire interne qui signale l'évolution du comportement du composant.

Nous considérons ici deux manières de maintenir la synchronisation des machines à états :
- une synchronisation par les événements, et
- une synchronisation par les transitions.

Synchronisation par les événements

Une machine à états évolue sous l'impulsion des événements qu'elle reçoit. Pour assurer l'évolution conjointe de deux machines à états similaires, il suffit donc de leur transmettre les mêmes séquences d'événements. Dans notre contexte, les requêtes de services sur le composant sans fil administré correspondent aux événements qui sont traités par sa machine à états interne. Ce sont donc ces mêmes messages qu'il faut transférer au système d'administration pour que la machine à états répliquée garde la cohérence avec celle du composant sans fil. C'est la première démarche que nous avons suivie (cf. [131, 130, 13]).

La figure 4.8 décrit sur un exemple les étapes d'une synchronisation par les événements. Dans cet exemple, le composant `Component` reçoit un événement `turnOff` à l'étape *a*. A l'étape *b*, l'événement est traité par la machine à états : la transition de l'état *On* à l'état *Off* est déclenchée et l'événement turnOff est transféré au système d'administration. Le transfert a lieu par un canal de communication sans fil, qui garantit la préservation de l'ordre des messages, placé entre le système sans fil et le système d'administration. Enfin, à l'étape *c*, l'événement `turnOff` est traité par la machine à états du composant répliqué ce qui le synchronise avec l'original.

Dans l'administration interne des composants WMX, nous avons vu que deux types d'informations devaient être capturées : les événements et les états de bas niveau[22]. Grâce à la

[22]Les états de bas niveau sont les attributs des objets qui constituent le composant comme défini précédemment (cf. section 4.2.2 p. 108).

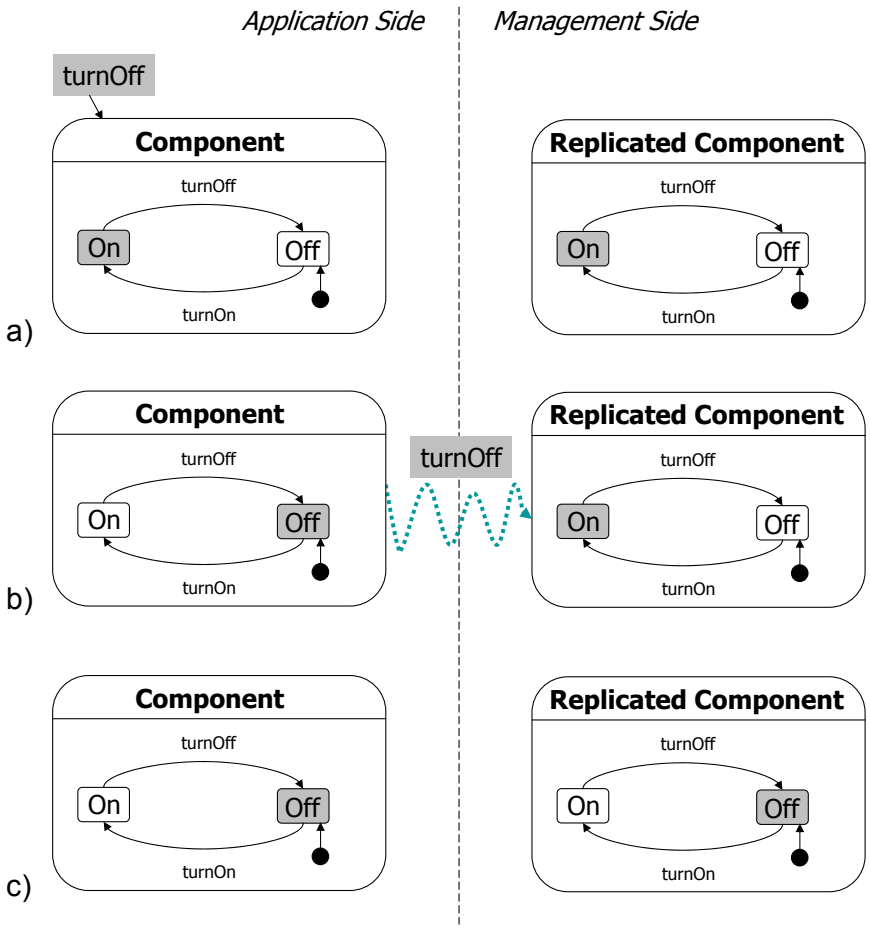


FIG. 4.8 – Les trois étapes de la synchronisation par les événements

réplication des machines à états, l'observation du comportement du composant peut s'effectuer seulement à partir des événements, ce qui diminue le flot des informations à collecter puisque les états de bas niveau ne sont pas transférés. L'observation du comportement ne s'effectue que sur les états abstraits définis par le *Statechart* du composant.

Cependant, cette approche n'est pas applicable lorsque le *Statechart* comprend des gardes faisant intervenir des états de bas niveau pour l'évaluation des transitions. En effet, puisque ces données ne sont pas transférées à la machine à états répliquée, celle-ci ne peut pas évaluer les mêmes gardes que la machine à états originale. Certaines transitions déclenchées dans le composant original ne sont donc pas reproduites dans sa réplique, l'exécution des machines à états diverges et le système d'administration produit une vue erronée du comportement du composant administré.

La figure 4.9 illustre ce problème.

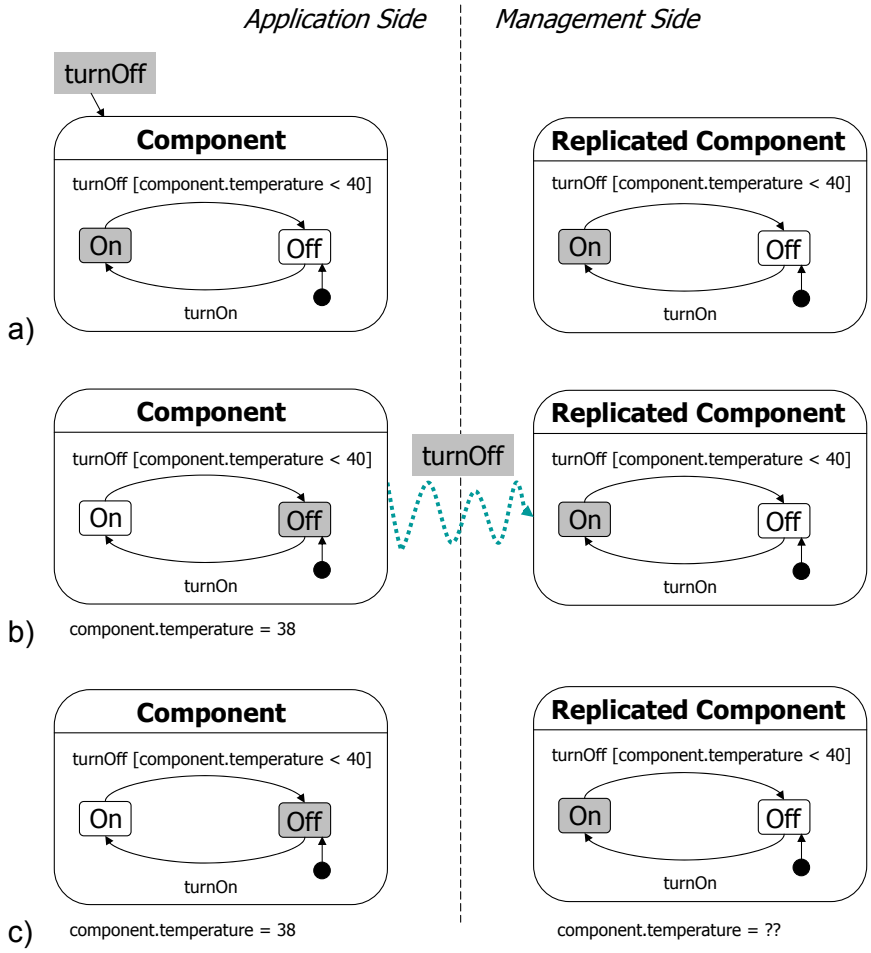


FIG. 4.9 – Problème de l'évaluation des gardes avec la synchronisation par les événements

Par rapport à l'exemple précédent, la transition de l'état *On* vers l'état *Off* est conditionnée par la valeur de l'attribut `component.temperature`. Après avoir reçu l'événement `turnOff` à

l'étape *a*, cet événement est traité par le composant sans fil à l'étape *b* : comme la garde de la transition est satisfaite, la machine à états transite vers l'état *Off*, et l'événement `turnOff` est transféré au système d'administration. A l'étape *c*, bien que l'événement `turnOff` soit le déclencheur de la transition vers l'état *Off*, cette transition ne peut être franchie car la machine à état répliquée ne peut évaluer l'attribut `component.temperature`. Les deux machines à états ne correspondent alors plus.

Synchronisation par les transitions

Dans l'approche précédente, nous avons vu que les événements ne sont pas une information suffisante pour permettre à la machine à états répliquée d'évaluer toutes les transitions. Afin d'éviter le problème de l'indétermination des transitions gardées, nous présentons maintenant une approche de synchronisation par les transitions.

Plutôt que d'évaluer deux fois un événement, dans le système sans fil puis dans le système d'administration, le résultat du *run-to-completion* de la machine à états du composant sans fil est transmis au système d'administration qui n'a plus qu'à positionner la machine à états répliquée en conséquence. C'est la démarche que nous adoptons maintenant dans notre solution d'administration (cf. [132, 133]).

La figure 4.10 montre comment la synchronisation par les transitions permet de résoudre le problème précédent.

Par rapport à l'exemple de la figure 4.9 qui considère le même composant à administrer, la machine à états du composant répliqué n'est pas la même. Elle ne comprend que les états de la machine originale. Après avoir reçu l'événement `turnOff` à l'étape *a*, le composant passe à l'état *Off* à l'étape *b*, car les conditions de déclenchement de la transition sont satisfaites. Cette même transition est alors transférée au composant répliqué qui l'exécute. Les deux machines à états sont à nouveau synchronisée à l'étape *c*.

Avec cette approche, les transitions entre les états sont construites à la volée dans la machine répliquée, ce qui permet de parcourir le graphe complet des états. La construction de ces transitions ne peut être initiée que par la machine à états originale. Les actions de contrôle sur le composant répliqué n'interviennent qu'indirectement sur la machine à états répliquée et sont d'abord opérées sur le composant administré. C'est ce que nous verrons dans les sections suivantes.

4.3.2 Capteurs et effecteurs externes

Afin de permettre l'administration distante de nos composants, ils doivent être équipés de capteurs et d'effecteurs externes. La figure 4.11 montre l'architecture détaillée du port d'administration externe (`Management Port`) présenté dans la structure d'un composant WMX en figure 4.5.

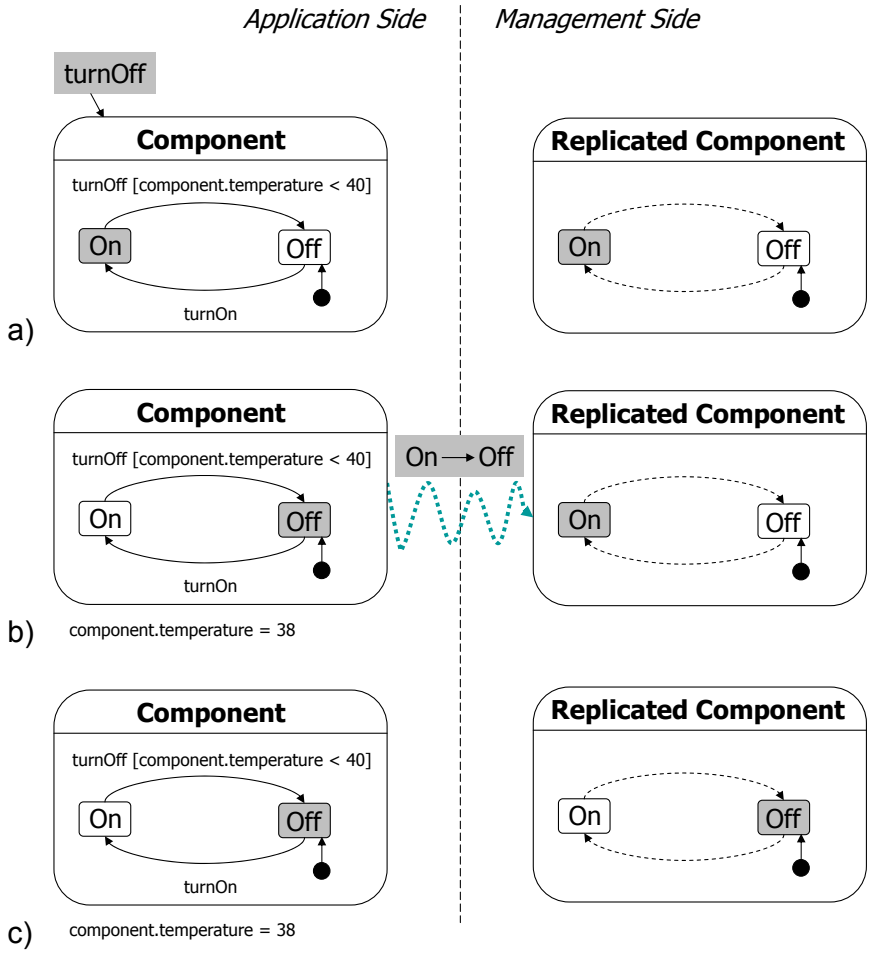


FIG. 4.10 – Les trois étapes de la synchronisation par les transitions

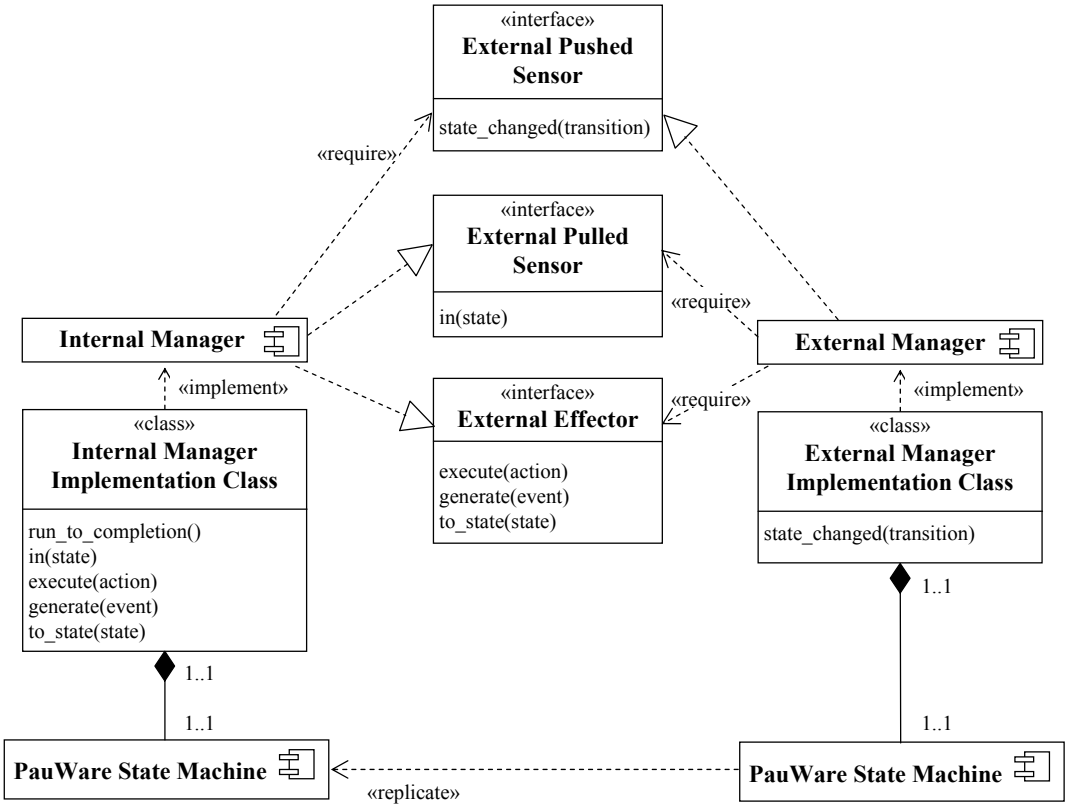


FIG. 4.11 – Architecture détaillée du port d'administration qui relie les gestionnaires interne et externe d'un composant WMX

L'instrumentation des composants WMX est réalisée par des interfaces placées sur le gestionnaire interne à qui une partie de l'administration du comportement des composants est déléguée (cf. section 4.2). Le gestionnaire interne exerce un premier contrôle sur le comportement du composant et seul le comportement abstrait qui résulte de ce pré-traitement à besoin d'être collecté par le gestionnaire externe.

Les interfaces qui permettent cette instrumentation sont au nombre de trois :

- `External Pushed Sensor`,
- `External Pulled Sensor`, et
- `External Effector`.

A l'instar de l'instrumentation interne, les deux capteurs externes suivent les modèles de collecte de données *push* et *pull*. Le capteur `External Pushed Sensor` utilise le modèle *push* pour notifier au système d'administration les changements d'état du composant. A chaque cycle *run-to-completion* de la machine à états qu'il contrôle, le gestionnaire interne communique les informations concernant des transitions effectuées grâce à la méthode `state_changed()`. Ces transitions sont alors appliquées par le gestionnaire externe qui contient la machine à états répliqué correspondante. L'échange de messages induit dans l'observation d'un événement est décrit par le diagramme de séquence de la figure 4.12.

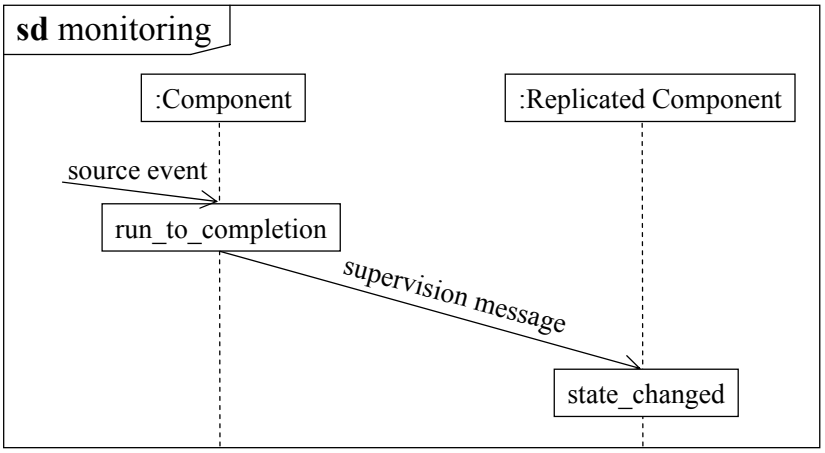


FIG. 4.12 – Echange de messages entre un composant et sa réplique lors de l'activité d'observation

Un second capteur, l'`External Pulled Sensor`, qui suit le modèle *pull* est également inclus. Le service fourni `in()` permet de vérifier que la machine à états se trouve bien dans un état donné. Cela est utile pour des besoins de re-synchronisation immédiat, autrement dit sans attendre le prochain traitement d'un cycle *run-to-completion*, après une perte de connexion par exemple.

Enfin, l'effecteur `External Effector` permet d'effectuer différents types de contrôle sur la machine à états du composant WMX comme nous allons le voir dans la section suivante.

4.3.3 Contrôle à distance

Dans l'activité de contrôle, le système d'administration ne communique pas directement avec le composant WMX mais passe par l'intermédiaire de son gestionnaire externe. Ainsi lorsque le gestionnaire externe reçoit une opération de contrôle, celle-ci est transférée au gestionnaire interne. Le gestionnaire interne exécute l'opération en question et l'applique sur la machine à états du composant. En retour, le gestionnaire externe reçoit les changements d'état grâce au mécanisme d'observation du capteur `External Pushed Sensor`. La machine à états répliquée du gestionnaire externe présente alors le résultat de l'opération de contrôle. Le diagramme de séquence de la figure 4.13 illustre la collaboration entre les gestionnaires interne et externe lors du traitement d'une opération de contrôle.

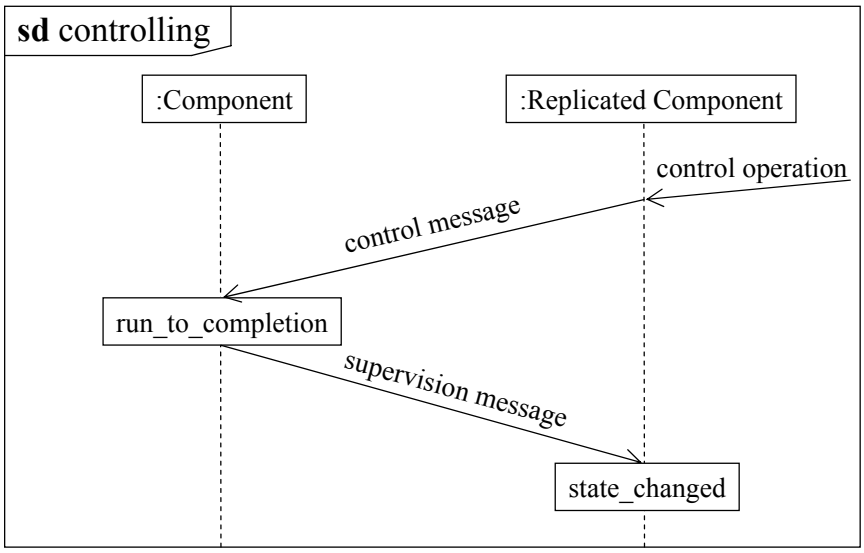


FIG. 4.13 – Echange de messages entre un composant et sa réplique lors de l'activité de contrôle

L'effecteur externe (`External Effector`) supporte trois types de contrôle :

- le contrôle par événement (service `generate()`),
- le contrôle par état (service `to_state()`), et
- le contrôle par action (service `execute()`).

Le gestionnaire externe réalise un contrôle par événement en envoyant au gestionnaire interne un événement qui correspond à une requête d'un service de l'interface de services fournis du composant. La requête du gestionnaire externe est traité comme pour tout client applicatif normal, c'est-à-dire que l'événement correspondant est évalué par le gestionnaire interne en fonction de la machine à états du composant. C'est un moyen de contourner l'interface de services fournis du composant. A la différence des autres clients, le gestionnaire externe peut également générer un événement interne, non disponible par l'interface de services fournis, mais néanmoins contrôlé via la machine à états.

Dans un contrôle par état, le gestionnaire externe ordonne au gestionnaire interne de positionner la machine à états du composant dans un état particulier défini par le modèle exécuté. Les contrôles induits par les transitions de la machine à états, autrement dit le mécanisme du *run-to-completion*, sont contournés et l'état courant de la machine à états est forcé dans le nouvel état. Cette opération permet de naviguer sur les états du *Statechart* comme si les graphes qui correspondent aux états mutuellement exclusif (XOR) étaient complets.

Enfin, un contrôle par action est effectué lorsque le gestionnaire externe ordonne au gestionnaire interne d'exécuter une action sur le composant métier sans que soit modifié l'état courant de la machine à états. C'est le type de contrôle de base que l'on retrouve le plus souvent dans les infrastructures d'administration comme JMX par exemple. La machine à états n'intervient pas.

4.4 Exemple d'une politique pour la gestion de compositions

Dans ce chapitre, nous avons vu comment le gestionnaire interne fournit au système d'administration une connaissance d'un plus haut niveau d'abstraction que les simples valeurs des attributs du composant administré. Par le biais des gestionnaires externes, il est ainsi possible d'envisager des politiques d'administration sur un assemblage de composants administrés qui n'auraient pas pu être prises en compte directement par les gestionnaires internes.

A titre d'exemple, nous présentons dans cette section une politique d'administration qui maintient à l'exécution la cohérence du comportement global de l'application par rapport à un type particulier de composition du comportement des composants, à savoir une composition de *Statecharts* selon la relation Tout-Partie [12].

4.4.1 Composition comportementale avec les *Statecharts* et la relation Tout-Partie

Nous avons vu que, dans l'ingénierie des composants logiciels, un système logiciel est perçu comme un assemblage de composants. L'objectif, qui est de favoriser la réutilisation du logiciel, amène souvent à la construction de nouvelles solutions qui combinent des composants développés en interne et d'autres déjà construits et acquis via un fournisseur. Cependant, la construction de systèmes à partir de pièces déjà existantes est connue pour être une tâche difficile [56].

Dans l'optique de faciliter la réutilisation des composants, Pazzi propose l'adoption des *Part-Whole Statecharts* (PWS) [121]. Les PWS s'inspirent de la relation Tout-Partie [89] afin d'associer des composants modélisés à partir de *Statecharts*, les « Parties », sans avoir besoin de définir leurs dépendances au sein de leurs comportements pour les faire collaborer. Les *Statecharts* de ces composants ne décrivent que leur propre comportement et ne sont pas utilisés pour contrôler le comportement d'autres composants. C'est le comportement du composant composite résultant

de la composition de « Parties », le « Tout », qui assure la coordination de ses différentes « Parties ». Le *Statechart* du « Tout » explicite alors la composition en présentant le comportement de l'assemblage à un plus haut niveau d'abstraction.

Dans sa proposition, les *Statecharts* qui représentent le comportement des composants « Parties », sont composés grâce à l'opérateur d'orthogonalité AND. Le résultat fournit le comportement de l'assemblage sous la forme d'un automate global contenant les *Statecharts* des composés dans des régions orthogonales différentes. Les communications entre les différents composants « Parties » sont cependant proscrites. Seuls sont autorisés les envois de signaux (qui correspondent à la notation `^signal` des *Statecharts*) avec le composant « Tout », dont le comportement est représenté dans une région supplémentaire, ajoutée au même automate global.

L'état de fonctionnement de l'assemblage est ainsi décrit deux fois : une première fois avec l'ensemble des états des « Parties » et une seconde avec les états du « Tout ». Les états du « Tout » correspondent en fait à des configurations particulières des états des « Parties », dont la distinction intervient dans une modélisation plus abstraite du comportement de l'assemblage.

Cependant, la correspondance entre ces deux niveaux d'états n'est pas assurée par la machine à états comme peuvent l'être états et sous-états dans la hiérarchie des états des *Statecharts*. Cette relation, qui entre dans la modélisation de l'assemblage, est créée artificiellement avec des signaux. Cela peut mener à une incohérence du *Statechart* du « Tout » par rapport aux *Statecharts* de ses « Parties » comme le montre l'exemple suivant.

Considérons un composant `TrafficLight`, qui implémente un feu tricolore, construit à partir de trois composants de type `Light`, *i.e.* des lampes, nommés `RedLight`, `YellowLight` et `GreenLight`. Ces composants font partie d'une relation Tout-Partie dans laquelle le composant `TrafficLight` est le composite et les composants `Light` sont les composés. Les composants `Light` ont le même comportement qui comprend deux états, *On* et *Off*, comme spécifié par le *Statechart* de la figure 4.14.

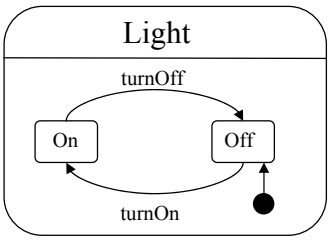


FIG. 4.14 – Comportement du composant `Light`

Le comportement du composant `TrafficLight` est représenté par le *Statechart* de la figure 4.15. Il comprend trois états principaux, *Red*, *Yellow* et *Green*, et un quatrième état `Start` qui permet d'initialiser la machine à états dans l'état *Red*. Quand une transition est déclenchée, le feu tricolore émet des signaux pour allumer ou éteindre ses lampes de sorte que seule la lampe dont le nom correspond à l'état atteint par la transition soit allumée.

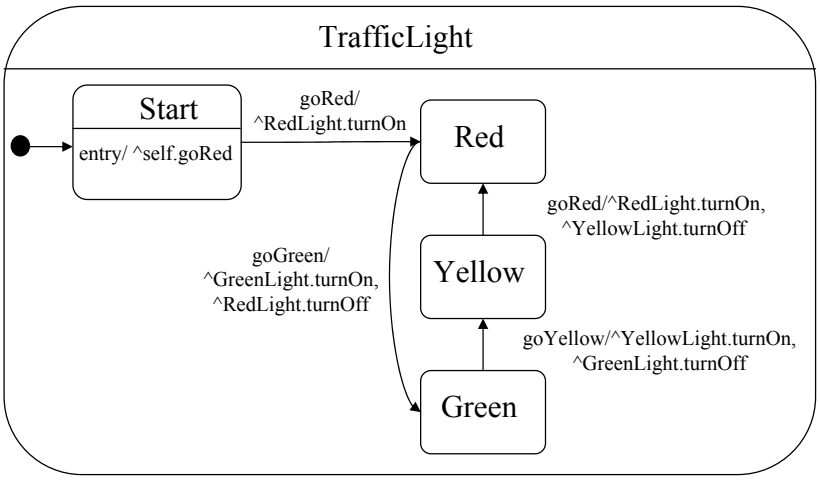


FIG. 4.15 – Comportement du composant `TrafficLight`

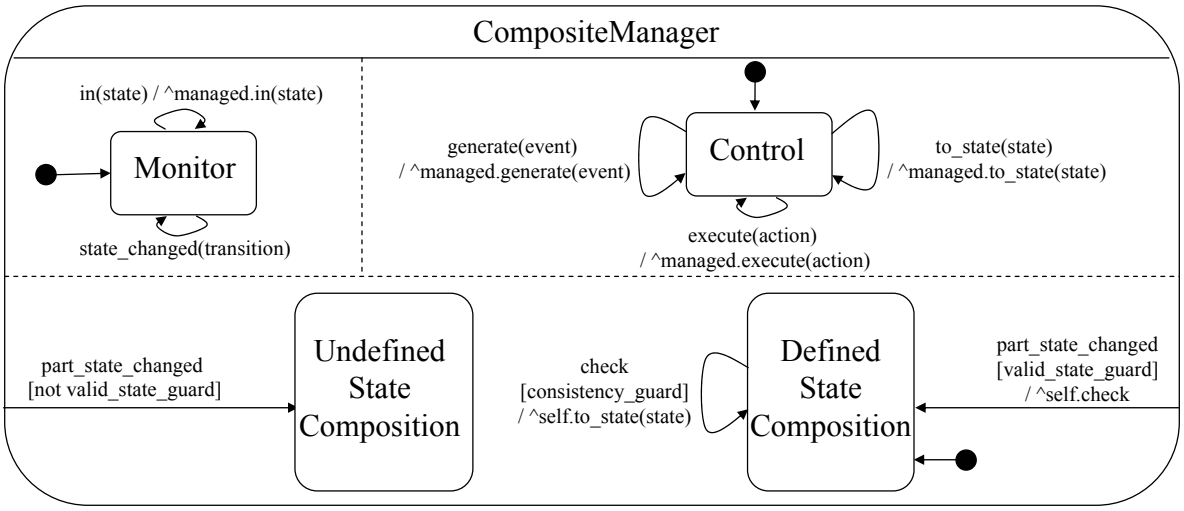
Spécifié à la manière de Pazzi, le système fonctionne bien tant que le contrôle des composés, *i.e.* les composants `Light`, ne provient que du composite, *i.e.* le composant `TrafficLight`. En effet, si un changement d'état survient sur l'un des composants `Light`, sans que le composant `TrafficLight` en soit à l'origine, l'état de fonctionnement du composite peut ne plus correspondre aux états de ses parties. Par exemple, si un événement `turnOn` est envoyé au composant `GreenLight` alors que le `TrafficLight` est dans l'état *Red*, `GreenLight` passe à l'état *On* et l'état de `TrafficLight` reste le même. Ce dernier ne décrit alors plus l'état de ses composés. Ceci est une illustration du problème précédemment évoqué.

Dans notre exemple, l'encapsulation des composants `Light` dans le composant `TrafficLight` pourrait être une solution au problème. Cependant, l'encapsulation n'est pas une caractéristique systématique de la relation Tout-Partie comme cela est précisé dans les travaux de formalisation de Barbier *et al.* [12] sur ce type de relation. De plus, l'encapsulation n'est pas une garantie sûre que les sous-composants ne seront pas accédés soit par malveillance à l'exécution, soit par négligence sur une erreur de conception. C'est le genre d'éléments imprévus qui peuvent par contre être détectés et corrigés par un système d'administration.

4.4.2 Une politique d'administration pour assurer la composition rigoureuse du comportement

Nous montrons ici comment le problème décrit dans l'exemple précédent est traité par notre système d'administration de composants logiciels. Pour cela le composant `TrafficLight` et les trois composants `Light` sont conçus comme des composants WMX exécutant les *Statecharts* des figures 4.14 et 4.15. Au niveau du système d'administration, nous proposons une solution générique, un *pattern*, pour concevoir des gestionnaires externes permettant de contrôler les composants dans une relation Tout-Partie. A la différence des composants WMX classiques, le fonctionnement des gestionnaires externes est différencié selon qu'ils administrent un composite

ou un composé. Les différents gestionnaires sont ainsi mis en relation afin de réunir leurs informations et mettre en œuvre des politiques d'administration sur l'ensemble de leurs composants WMX. Le comportement d'un gestionnaire de composite est décrit par la figure 4.16 et celui d'un gestionnaire de composé par la figure 4.17. Seules les gardes sont spécifiques à l'exemple du `TrafficLight`.



valid_state_guard : (RedLight.in(On) ∧ YellowLight.in(Off) ∧ GreenLight.in(Off))

∨ (RedLight.in(Off) ∧ YellowLight.in(On) ∧ GreenLight.in(Off))

∨ (RedLight.in(Off) ∧ YellowLight.in(Off) ∧ GreenLight.in(On))

consistency_guard : (state = Red) ⇒ (managed.in(Red) ⇒ (RedLight.in(On) ∧ YellowLight.in(Off) ∧ GreenLight.in(Off))

∨ (state = Yellow) ⇒ (managed.in(Yellow) ⇒ (RedLight.in(Off) ∧ YellowLight.in(On) ∧ GreenLight.in(Off))

∨ (state = Green) ⇒ (managed.in(Green) ⇒ (RedLight.in(Off) ∧ YellowLight.in(Off) ∧ GreenLight.in(On))

FIG. 4.16 – Comportement du `CompositeManager`

Le principe de la politique que nous proposons pour gérer le problème que nous avons décrit, consiste à spécifier les états du composite comme des états abstraits qui appartiennent à un sous-ensemble du produit cartésien des états des composés. Dans l'exemple du feu tricolore, le composant `TrafficLight` est composé de trois composants `Light` et chacun de ces composants comprend deux états. Ainsi, le produit cartésien des états des composés donne 2^3 états et seulement trois états du `TrafficLight` y correspondent (*Red*, *Yellow* et *Green*). Parmi les états possibles, ceux qui décrivent le fait que plus d'une lampe est allumée ou alors qu'aucune ne l'est, sont indéfinis pour le `TrafficLight`. Le tableau 4.1 illustre cette situation.

Ainsi, la composition entre le composant `TrafficLight` et ses sous-composants `Light` peut être caractérisée par deux états, *Defined* ou *Undefined*, selon que les états des composants `Light` décrivent ou non un état valide pour le `TrafficLight`. Cela est vérifié par la garde `valid_state_guard` de la figure 4.16. L'état *Undefined* indique au système d'administration que l'assemblage est dans un état qui n'a pas été spécifié. Il doit alors être géré manuellement en repositionnant les composants dans un état cohérent. Par contre, si les états des composés

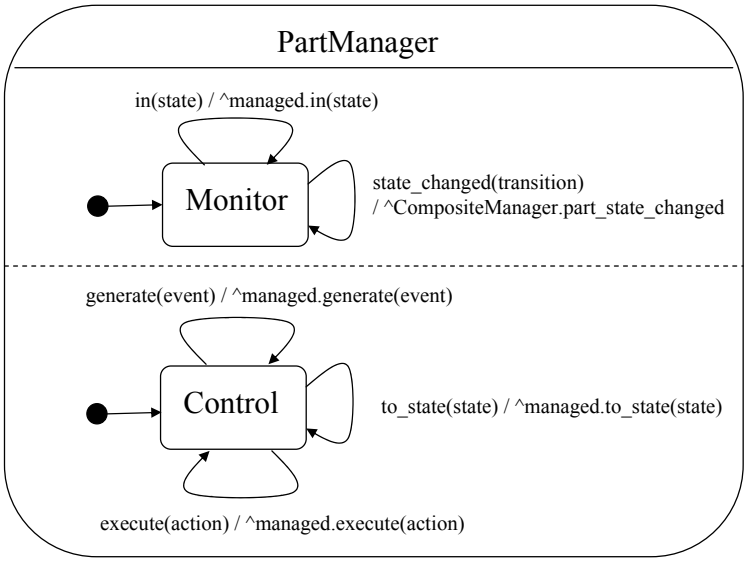


FIG. 4.17 – Comportement du `PartManager`

Composés	**Combinaisons d'états valides**		
RedLight	On	Off	Off
YellowLight	Off	On	Off
GreenLight	Off	Off	On
TrafficLight	Red	Yellow	Green
Composite	**Etats correspondants**		

TAB. 4.1 – Correspondance des états entre composés et composite

correspondent à un état qui est défini par la composition, le gestionnaire du composite vérifie si l'état du composant est bien dans cet état. S'il ne l'est pas, le gestionnaire repositionne automatiquement la machine à états du composite dans l'état correspondant. Cette vérification est effectuée par la garde `consistency_guard` de la figure 4.16.

4.5 Synthèse

Dans ce chapitre, nous avons décrit l'architecture que nous avons conçue pour administrer les composants WMX qui sont des composants déployables sur des systèmes sans fil et administrables à distance. L'une des particularités majeures de cette architecture est qu'elle effectue l'administration du comportement des composants sur deux niveaux.

Au niveau interne, nous avons vu comment le gestionnaire interne d'un composant WMX contrôle la machine à états qui exécute l'état de fonctionnement du composant. Le gestionnaire interne est l'élément central de notre architecture puisqu'il permet de faire le lien entre la partie métier du composant et le système d'administration distant.

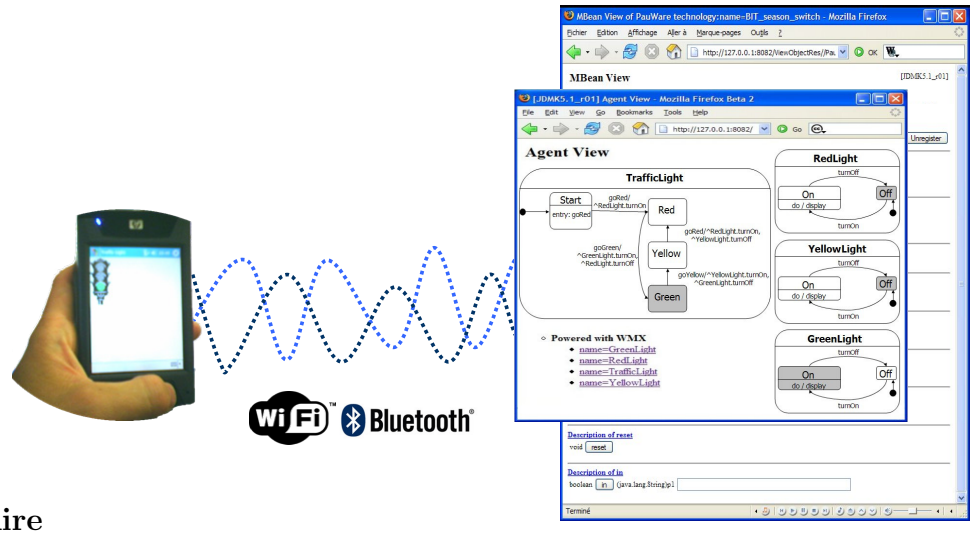
Au niveau externe, nous avons montré comment la machine à états des composants est répliquée et synchronisée dans le système d'administration. Les gestionnaires internes agissent ainsi comme un filtre sur les informations d'administration à collecter pour ne transmettre que l'état de fonctionnement aux gestionnaires externes. Ces derniers peuvent alors collaborer pour appliquer des politiques d'administration basées sur une vue abstraite du comportement des composants WMX.

Nous avons ainsi pu mettre en œuvre sur un exemple une politique d'administration permettant de gérer la cohérence des états des composants dans une relation Tout-Partie. Cette politique permet de détecter les états des composants qui correspondent à des configurations invalides pour la composition et permet de resynchroniser automatiquement leurs états dans les configurations valides.

Chapitre 5

Wireless Management eXtensions

Une implémentation Java pour l'administration de composants logiciels sur systèmes sans fil



Sommaire

5.1 L'implémentation de la réflexion dans PauWare.Velcro : un pré-requis 126

5.1.1 Utilisation de la réflexion Java dans PauWare 126

5.1.2 Implémentation d'un mécanisme remplaçant la réflexion pour Java ME 127

5.1.3 Génération de code 131

5.2 Mise en œuvre de l'architecture avec des technologies Java 131

5.2.1 Implémentation des composants WMX 132

5.2.2 Ports de communication sans fil et système d'administration à distance 134

5.3 Visualisation du comportement 135

5.3.1 La console JMX 135

5.3.2 L'outil PauWareView 137

5.4 Etude de performance 137

5.4.1 Elaboration du banc d'essai et mesures 139

5.4.2 Analyse des résultats 140

L'approche et l'architecture que nous avons présentées dans les précédents chapitres, ont été mises en œuvre, et ont à cet égard présidé à la création de la technologie *Wireless Management eXtensions* (WMX)[23]. WMX est un ensemble de librairies et d'outils qui associent les différentes plates-formes Java pour fournir un ensemble homogène couvrant notre problématique de l'administration du comportement des composants logiciels aussi bien au niveau du système sans fil qu'au niveau du système d'administration. Ainsi, la plate-forme Java ME est utilisée pour réaliser les composants administrables dans les systèmes sans fil, et la plate-forme Java SE est utilisée avec JMX pour le système d'administration distant.

Dans ce chapitre, nous nous intéressons à la mise en œuvre de WMX. Nous commençons par détailler le mécanisme de réflexion implémenté dans PauWare.Velcro qui a permis l'utilisation des machines à états PauWare dans les composants WMX déployés sous Java ME. Puis nous décrivons la mise en œuvre de l'architecture d'administration elle-même et les outils de visualisation du comportement des composants WMX. Enfin, nous concluons ce chapitre par une étude évaluant les performances de WMX, afin de mesurer d'éventuels *overheads*[24] induits par notre solution d'administration.

5.1 L'implémentation de la réflexion dans PauWare.Velcro : un pré-requis

Le traitement des actions dans la librairie PauWare utilise les mécanismes de réflexion du langage Java (paquetage `java.lang.reflect`). Or ce mécanisme n'est pas disponible dans CLDC, la configuration qui équipe à ce jour la plupart des appareils sans fil supportant Java ME. L'implémentation d'un mécanisme permettant de pallier ce manque s'impose donc comme un pré-requis à la réalisation de WMX.

5.1.1 Utilisation de la réflexion Java dans PauWare

Java, comme d'autres langages de programmation (SmallTalk, CLOS, ou Python par exemple), offre des mécanismes de réflexion. Ces mécanismes permettent de traiter au sein d'un programme les instructions de ce programme comme des données : on parle alors de métaprogrammation. L'API de réflexion Java fournit ainsi un ensemble de classes permettant de manipuler principalement les concepts orientés objet tels que la classe avec `java.lang.Class`, les méthodes avec `java.lang.reflect.Method` ou les attributs avec `java.lang.reflect.Field`.

En utilisant la réflexion Java, il est par exemple possible d'effectuer un appel de méthode sur un objet à partir d'une chaîne de caractères contenant le nom de la méthode à appeler. L'exemple du listing 5.1 illustre cette utilisation pour invoquer les méthodes `aFirstMethod()` et `aSecondMethod()` sur le même objet `object`.

[23] `http://wmx.fromeo.fr`
[24] Surcoûts en temps de calcul.

```
Method method = null;
String methodToInvoke = "aFirstMethod";

method = object.getClass().getMethod(methodToInvoke, null);
method.invoke(object, null);

methodToInvoke = "aSecondMethod";
method = object.getClass().getMethod(methodToInvoke, null);
method.invoke(object, null);
```

List. 5.1 – Utilisation de la réflexion pour l'invocation de méthodes

Grâce à la méthode `invoke()` de l'API de réflexion Java, le choix de la méthode à exécuter sur un objet peut être décidé à l'exécution et non plus à la compilation par un appel de méthode traditionnel. Cette possibilité est à la base du principe d'exécution des actions et de l'évaluation des gardes dans la machine à états PauWare. Pour permettre de définir les transitions de la machine à états de façon générique, les actions et les gardes sont enregistrées par des chaînes de caractères via la méthode `fires()` (cf. listing 3.1 p. 90). Lors d'un cycle *run-to-completion*, les méthodes correspondantes sont alors appelées par réflexion comme nous venons de le voir.

Si l'API de réflexion est maintenant disponible sur la plate-forme Java standard depuis la version 1.2, cela n'est pas le cas pour la plate-forme destinée aux systèmes sans fil, Java ME. En effet, CLDC qui est la configuration la plus utilisée de Java ME sur les appareils sans fil n'inclut pas cette API. Parmi les différentes modifications nécessaires à l'utilisation de la machine à états PauWare sur des systèmes sans fil, nous avons ainsi dû mettre en œuvre dans PauWare.Velcro un mécanisme pour pallier l'absence de réflexion dans CLDC. Par conséquent, comme CLDC est le plus petit sous-ensemble de la plate-forme Java, PauWare.Velcro peut fonctionner sur le plus grand nombre d'appareils possible.

5.1.2 Implémentation d'un mécanisme remplaçant la réflexion pour Java ME

La réflexion dans PauWare est utilisée pour rendre la librairie la plus générique possible. Le but est de faciliter le travail de l'utilisateur, qui n'a alors besoin de donner qu'un minimum d'informations pour construire une machine à états. Bien que connu au moment du développement, le reste des informations est retrouvé par réflexion à l'exécution. Le principe de PauWare.Velcro s'appuie sur le fait que les informations manquantes sont préalablement connues pour remplacer la réflexion qui n'existe pas dans Java ME (CLDC). PauWare.Velcro a été conçu de sorte que l'utilisateur puisse préciser ces informations dès le développement de sa machine à états.

Pour cela, l'interface `VelcroExecutor` permet de déléguer au développeur d'un composant PauWare.Velcro la recherche des méthodes permettant à la machine à états d'exécuter ses ac-

tions et d'évaluer ses gardes[25]. Le listing 5.2 présente l'implémentation Velcro du composant `PauWareComponent`.

Pour être contrôlé par sa machine à états, le composant doit donc implémenter l'interface `VelcroExecutor` et sa méthode `execute()`. La méthode `execute()` permet au développeur de décrire les noms des actions et des gardes avec leurs méthodes correspondantes grâce à un test sur le paramètre `action`. Les envois de signaux sur le composant lui-même sont traités comme des actions et doivent donc également figurer dans la méthode `execute()`.

Dans notre exemple, le composant `PauWareComponent` ne communique pas avec d'autres composants. Pour permettre la communication entre des composants PauWare via leurs machines à états, tous les événements des composants doivent également être référencés dans la méthode `execute()`. Ainsi, la machine à états d'un composant peut provoquer l'exécution d'un événement sur un autre composant.

Dans certains cas, il peut être nécessaire que la machine à états d'un composant contrôle un objet externe au composant sur lequel l'implémentation de l'interface `VelcroExecutor` est trop lourde à imposer. Cela peut être le cas par exemple d'une interface graphique. PauWare.Velcro fournit pour cela une seconde interface, l'interface `VelcroActionExecutor`. Par rapport à `VelcroExecutor`, la méthode `execute()` de l'interface `VelcroActionExecutor` possède un paramètre supplémentaire désignant l'objet sur lequel l'action doit être effectuée. Le listing 5.3 présente l'utilisation de cette interface sur l'exemple d'un composant `Light` contrôlant un objet de type `LightGUI`.

Le style de programmation de PauWare.Velcro est certes moins élégant que celui de PauWare, mais cette nouvelle version permet de résoudre l'absence de réflexion dans Java ME (CLDC). Ce mécanisme est également applicables sur les autres plate-formes Java, ce qui fournit un moyen pour éviter l'utilisation de la réflexion Java, qui est connue pour être pénalisante à l'exécution. Toutefois, les performances de PauWare.Velcro ne sont pas meilleures que celles de PauWare, comme nous le verrons en section 5.4. Des optimisations peuvent être apportées à la librairie en utilisant par exemple les mécanismes de *SmartReflection* [28]. *SmartReflection* améliore sensiblement les temps d'exécution de l'invocation de méthodes par réflexion, en exploitant des particularités d'implémentation de la JVM considérée. Le code est alors plus performant mais au prix d'une moindre portabilité.

Les avantages et inconvénients des différents styles de programmation des machines à états PauWare ont scindé la librairie en deux branches : *Composytor* qui utilise la réflexion et *Velcro* qui utilise le mécanisme que nous venons de présenter. La figure 5.1 présente la nouvelle architecture de PauWare qui permet de gérer l'évolution conjointe de ces deux branches grâce aux classes abstraites `AbstractStatechart` et `AbstractStatechart_monitor`.

[25]Précisons que les méthodes d'actions et de gardes ne doivent pas être appelées directement car seule la machine à états est capable de décider du bon moment pour effectuer l'appel. En effet, l'entrelacement des différentes méthodes n'est pas forcément toujours le même mais suit le comportement résultant de l'automate décrit.

```
1  public class PauWareComponent implements VelcroExecutor {
2
3      /* ... */
4
5      public Object execute(String action, Object[] args)
6                                          throws Throwable {
7          Object result = null;
8          if(action == null) return result;
9
10         /* UML actions */
11         else if(action.equals("action_s10"))
12             this.action_s10();
13         else if(action.equals("action_s11"))
14             this.action_s11();
15         else if(action.equals("action_s12"))
16             this.action_s12();
17         else if(action.equals("action_s2"))
18             this.action_s2();
19         else if(action.equals("action_s3"))
20             this.action_s3();
21
22         /* UML guards */
23         else if(action.equals("guard_s10s11"))
24             this.guard_s10s11();
25         else if(action.equals("guard_s10s12"))
26             this.guard_s10s12();
27
28         /* UML self-sent events */
29         else if(action.equals("event_s3"))
30             this.event_s3();
31
32         return result;
33     }
34 }
```

List. 5.2 – Utilisation de `VelcroExecutor` dans la version Velcro du composant `PauWareComponent` (cf. figure 3.16 p. 97)

```
public class Light implements VelcroActionExecutor {

    /* ... */

    public Object execute(Object object,
                          String action,
                          Object[] args)
                                        throws Throwable {
        Object result = null;
        if(action == null) return result;

        /* External actions */
        else if(action.equals("displayLightOn"))
            ((LightGUI)object).displayLightOn();
        else if(action.equals("displayLightOff"))
            ((LightGUI)object).displayLightOff();

        return result;
    }
}
```

List. 5.3 – Utilisation de `VelcroActionExecutor` dans la version Velcro du composant `Light` (cf. figure 3.11 p. 91)

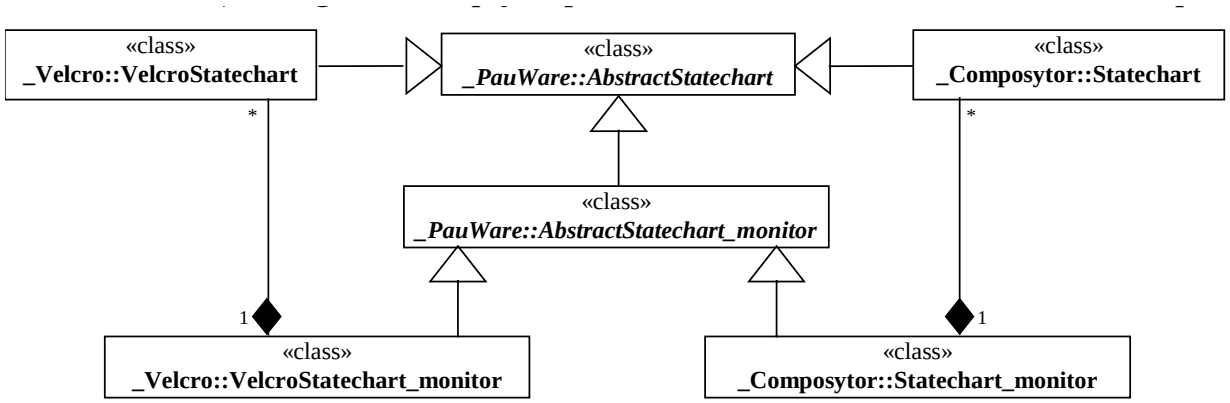


Fig. 5.1 – Architecture de PauWare pour Java ME et Java SE (extrait de [11])

5.1.3 Génération de code

Rappelons ici que l'API de PauWare permet de construire des modèles exécutables conformes à la notation et à la sémantique d'UML. Ces modèles peuvent être obtenus en utilisant directement et manuellement l'API comme nous l'avons fait ici. Cependant, dans le cadre d'une approche dirigée par les modèles, il est préférable de passer par des modèles intermédiaires réutilisables dans d'autres environnements de développement et à partir desquels le code peut être généré.

Pour intégrer la technologie PauWare dans un processus complet de développement dirigé par les modèles, l'outil XMI2PauWare a été développé. Cet outil est un générateur de code qui permet de produire le code de la machine à états PauWare à partir de modèles au format XMI (*XML Metadata Interchange*) [111]. XMI est un standard défini par l'OMG, qui s'appuie sur XML pour l'échange d'informations de métadonnées UML. Aujourd'hui, la plupart des outils de modélisation graphique pour UML propose d'exporter les modèles créés dans ce format. Cela permet de construire des générateurs de code indépendants des outils de modélisation. Ainsi, grâce à l'outil XMI2PauWare, PauWare fournit une solution complète de développement dirigé par les modèles en quatre étapes :

1. création du modèle de comportement avec un outil de modélisation graphique,
2. export du modèle au format XMI,
3. génération du code de la machine à états PauWare avec XMI2PauWare,
4. implémentation des méthodes qui correspondent aux actions et aux gardes de la machine à états.

5.2 Mise en œuvre de l'architecture avec des technologies Java

Dans WMX, les gestionnaires internes et externes de notre architecture sont construits au dessus de la librairie Java PauWare. Pour administrer les composants WMX, les gestionnaires internes utilisent la version Java ME, Velcro, et les gestionnaires externes utilisent la version Java SE, Composytor.

Les communications entre les composants WMX et le système d'administration sont délégués à des adaptateurs spécifiques permettant de supporter différentes technologies sans fil.

Le système d'administration global de WMX repose sur le standard d'administration JMX, ce qui rend les composants WMX compatibles avec d'autres solutions basées sur ce standard.

Dans cette section, nous détaillons dans un premier temps l'implémentation des composants WMX puis dans un second temps nous détaillons l'implémentation du système d'administration distant.

5.2.1 Implémentation des composants WMX

WMX permet d'implémenter des composants administrables et déployables sur des systèmes sans fil (supportant Java ME) : les composants WMX. Dans sa librairie, WMX fournit les constructions nécessaires pour faciliter l'implémentation des composants WMX avec des machines à états PauWare administrables.

D'un point de vue pratique, un composant WMX est construit comme une extension de la classe abstraite `WMX_component` fournie par la librairie WMX. Son comportement est ensuite exprimé sur un modèle similaire aux composants PauWare en utilisant la librarie PauWare.Velcro. Le listing 5.4 correspond au code source d'un composant WMX `Light` (par souci de clarté, une partie du code est éludée).

La seule différence entre la réalisation d'un composant WMX et celle d'un composant PauWare réside dans la façon de créer la machine à états du composant. Cette création ne s'effectue plus directement par l'instanciation des classes `VelcroStatechart` et `VelcroStatechart_monitor` mais par l'utilisation des méthodes `createWMXStatechart()` et `createWMXStatechart_monitor()`. Ces méthodes proviennent de la classe abstraite `WMX_component` qu'un composant WMX étend pour être administrable.

Ainsi, la majorité du code nécessaire à l'administration est déjà implémentée par la classe abstraite `WMX_component`. Par exemple, pour suivre l'évolution de la machine à états du composant, la classe `WMX_component` implémente l'interface `Statechart_monitor_listener` ainsi que les méthodes correspondantes (cf. figure 5.2) :

```
1 abstract public class WMX_component implements VelcroExecutor,
2     Statechart_monitor_listener {/* ... */}
```

Le composant WMX peut alors être enregistré sur l'`AbstractStatechart_monitor` défini par le développeur grâce à la méthode. Cet enregistrement est effectué automatiquement à la création de la machine à états du composant via la méthode `createWMXStatechart_monitor()` de la classe `WMX_component`.

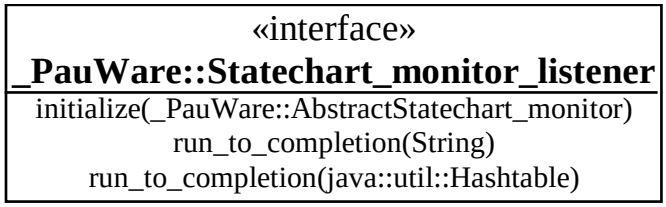


Fig. 5.2 – Interface `Statechart_monitor_listener` (extrait de [11])

La classe `WMX_component` contient également les méthodes `addWirelessPort()` et `start_management()` qui permettent respectivement d'ajouter un port de communication sans fil et de

```
1  public class Light extends WMX_component {
2      protected AbstractStatechart _On;
3      protected AbstractStatechart _Off;
4      protected AbstractStatechart_monitor _Light;
5      public Light() throws Statechart_exception {
6          init_behavior();
7      }
8      public void init_behavior() throws Statechart_exception {
9          // init states
10         _On = createWMXStatechart("On");
11         _On.entryAction(this,"displayLightOn");
12         _Off = createWMXStatechart("Off").inputState();
13         _Off.entryAction(this,"displayLightOff");
14     }
15     public void start() throws Statechart_exception {
16         // init state machine
17         _Light = createWMXStatechart_monitor(_On.xor(_Off),
18                                              "Light");
19         // init transitions
20         _Light.fires("turnOn",_Off,_On);
21         _Light.fires("turnOff",_On,_Off);
22     }
23     public void turnOn() throws Statechart_exception {
24         _Light.run_to_completion("turnOn");
25     }
26     public void turnOff() throws Statechart_exception {
27         _Light.run_to_completion("turnOff");
28     }
29     public Object execute(String action, Object[] args)
30                             throws Statechart_exception {
31         /* ... Velcro programming style ...*/
32     }
33     public void displayLightOn() {
34         /* ... */
35     }
36     public void displayLightOff() {
37         /* ... */
38     }
39 }
```

List. 5.4 – Implémentation WMX du componsant `Light` (cf. figure 3.11 p. 91)

```
1  public void state_changed(AbstractStatechart[] states)
2      throws Statechart_exception {
3      /* registration of dynamic transitions */
4      for(int i=0; i<states.length; i+=2) {
5          _statechart_monitor.fires(states[i],
6                                    states[i+1]);
7      }
8      /* update the state machine accordingly */
9      _statechart_monitor.run_to_completion();
10 }
```

LIST. 5.5 – Méthode `state_changed()` de la classe abstraite `WMX_Manager`

démarrer, par ce port, la transmission des informations d'administration au système d'administration distant.

5.2.2 Ports de communication sans fil et système d'administration à distance

Tout comme la classe `WMX_component` est dédiée au développement d'un composant WMX, la classe abstraite `WMX_manager` contient les constructions nécessaires pour implémenter des gestionnaires externes. La machine à états du gestionnaire externe d'un composant WMX est construite de manière symétrique à celle du composant administré grâce aux méthodes `createManagedStatechart()` et `createManagedStatechart_monitor()` de la classe `WMX_manager`. Les autres éléments de la machine à états, c'est-à-dire les transitions, les événements, et les actions, ne sont pas définis dans le gestionnaire externe. La machine à états du gestionnaire externe est synchronisée sur la machine à états interne au composant WMX grâce à la méthode `state_changed()` de la classe `WMX_manager`. Le code source de cette méthode est détaillé dans le listing 5.5.

A chaque cycle *run-to-completion* de la machine à états du composant WMX, le gestionnaire externe reçoit l'ensemble des couples (*état d'origine*,*état d'arrivé*) qui correspondent aux transitions déclenchées. A partir de ces couples, la méthode `state_changed()` a pour fonction de recréer dynamiquement et temporairement ces transitions (seulement le changement d'état) et de les activer sur la machine à états du gestionnaire externe.

Ces informations provenant des composants WMX sont reçues par les gestionnaires externes via des adaptateurs sans fil. La classe `WMX_manager` fournit les même méthodes que la classe `WMX_component`, `addWirelessPort()` et `start_management()`, pour ajouter et actionner ces ports de communication. Pour que la communication sans fil puisse fonctionner, le composant WMX et son gestionnaire doivent utiliser des ports de communication compatibles. Notre implémentation propose actuellement deux types d'adaptateurs fonctionnant sur des réseaux sans fil : un premier basé sur le connecteur réseau générique de Java ME, `javax.microedition.io.Connector`, qui peut être utilisé sur des réseaux Wifi, Bluetooth ou encore IrDA, et un

autre basé sur l'extension WMA (*Wireless Message API*) proposée dans le paquetage `javax.wireless.messaging` et qui fonctionne avec le service de messagerie SMS (*Short Message Service*).

Enfin, les gestionnaires externes sont implémentés comme des MBeans afin d'être accessibles via JMX, le standard d'administration Java. Le diagramme de la figure 5.3 montre l'architecture correspondante pour le gestionnaire du composant `Light`.

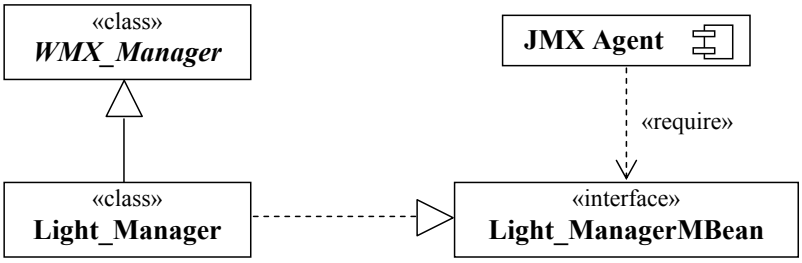


FIG. 5.3 – Implémentation du gestionnaire du composant `Light`

5.3 Visualisation du comportement

Le système d'administration de WMX est compatible avec JMX grâce aux interfaces MBeans des gestionnaires externes. L'administration des composants WMX peut ainsi profiter des outils supportant ce standard.

Dans cette section, nous présentons le principe de l'administration des composants WMX via la console JMX. Puis nous montrons sur l'exemple du `TrafficLight` (cf. 4.4.1 p. 120), les possibilités d'observation du comportement des composants avec l'outil de visualisation de machines à états PauWareView.

5.3.1 La console JMX

Nous avons vu au chapitre 2 (cf. section 2.3.1 p. 47) qu'un serveur JMX peut être relié à une console d'administration distante permettant de manipuler les MBeans enregistrés sur le serveur. Les gestionnaires WMX étant conçus comme des MBeans, c'est le moyen que nous utilisons pour accéder aux fonctionnalités qu'ils offrent pour administrer les composants WMX. La figure 5.4 décrit l'architecture globale de l'administration d'un composant WMX depuis le système sans fil jusqu'à la console JMX.

Nous utilisons deux types de consoles d'administration JMX. La première utilise l'adaptateur JDMK HTML qui permet d'envoyer les informations d'administration sous forme de pages HTML. La console consiste alors en un simple navigateur Web, ce qui évite l'installation d'un logiciel particulier sur la station de travail utilisée pour l'administration.

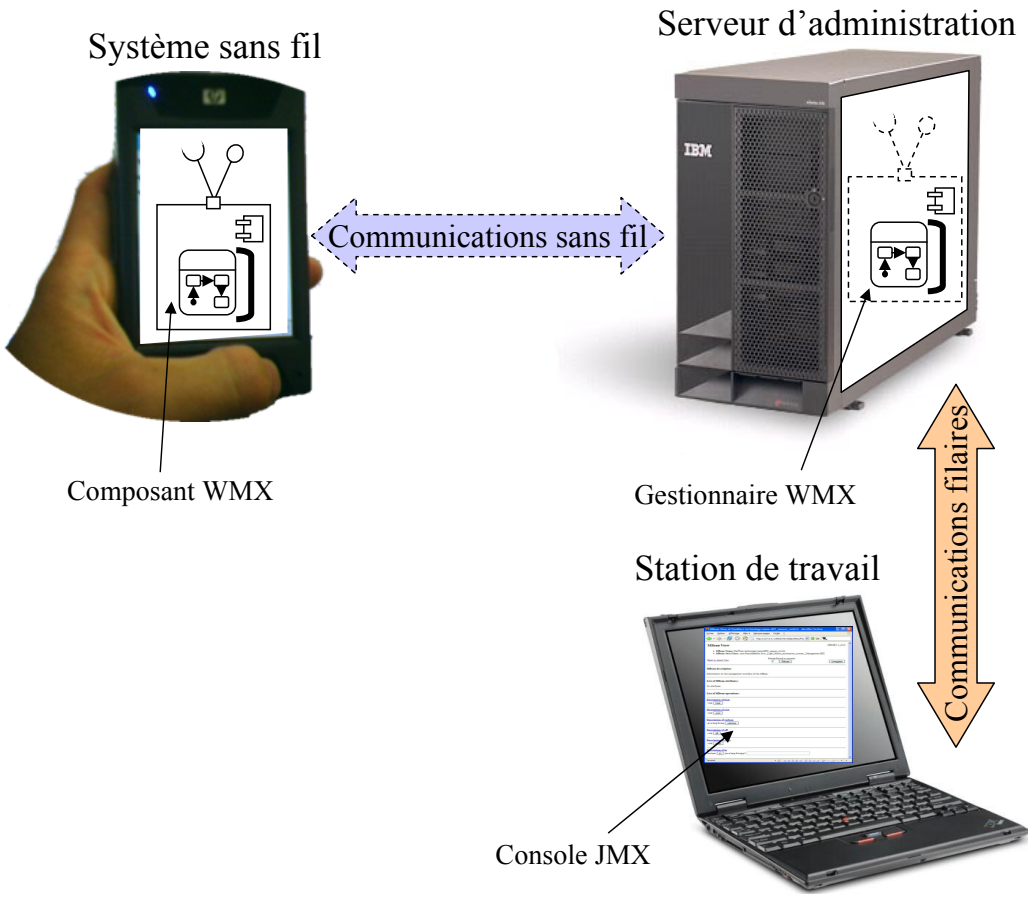


FIG. 5.4 – Administration d'un composant WMX via la console JMX

La seconde, la JConsole, utilise un adaptateur RMI pour se connecter au serveur JMX. La JConsole est l'outil fourni par SUN pour l'administration par JMX de sa machine virtuelle Java (la JVM). C'est une application Java graphique plus élaborée que la console précédente et qui permet notamment de visualiser les données des MBeans à administrer grâce à des courbes graphiques. Cependant, pour visualiser graphiquement le comportement des composants WMX, l'utilisation d'outils supplémentaires s'impose.

5.3.2 L'outil PauWareView

PauWareView est un outil de visualisation graphique permettant d'observer l'exécution des machines à états PauWare. Ce complément à la technologie PauWare a été développé par Cyril Ballagny et Philippe Carvalho dans le cadre du Master *Technologies de l'Internet*[26] de l'Université de Pau et des Pays l'Adour. L'implémentation de PauWareView est basée sur la librairie *opensource* OpenJGraph[27] qui permet de créer et de manipuler des graphes en Java.

Bien que les mêmes observations peuvent être effectuées via une console JMX, cet outil apporte un confort supplémentaire pour l'administrateur humain. Nous l'utilisons dans WMX au niveau du système d'administration pour visualiser la machine à états des gestionnaires externes. La figure 5.5 montre l'utilisation conjointe de PauWareView avec une console JMX pour administrer le composant WMX `TrafficLight` (cf. 4.4.1 p. 120).

En a), l'interface graphique du PauWareView affiche un diagramme représentant la machine à états du `TrafficLight`, dans lequel l'état courant, *Red*, est représenté en grisé. Un navigateur Web permet d'accéder à la console JMX en b). La page du navigateur liste les différentes opérations d'administration disponibles sur le composant, parmi lesquelles se trouve l'opération `goGreen()` qui permet de contrôler le composant en transmettant l'événement du même nom à sa machine à états. L'exécution de cette opération est déclenchée en b) et confirmée en c). A la réception des informations d'administration provenant du composant, la vue du PauWareView est mise à jour en d). L'état *Red* n'est plus grisé car l'état courant a changé : la machine à états se trouve maintenant dans l'état *Green*.

5.4 Etude de performance

D'un point de vue qualitatif, les performances de WMX nous ont permis de déployer sur des pockets PC des applications administrables à distance, ce qui était l'objectif visé. Dans le but de minimiser la durée de nos travaux, nos expérimentations ont porté principalement sur l'exécution à partir d'un HP iPAQ hx4700 de l'application *Traffic Light* présentée au chapitre 4 et son administration via un réseau Wifi. Des applications plus complexes comme le *Home Automation System* décrit dans le manuel d'utilisation de PauWare [11] sont encore à l'étude. Néanmoins, à l'heure actuelle, l'implémentation PauWare.Velcro de ces applications a pu être

[26] `http://masterti.univ-pau.fr`
[27] `http://openjgraph.sourceforge.net`

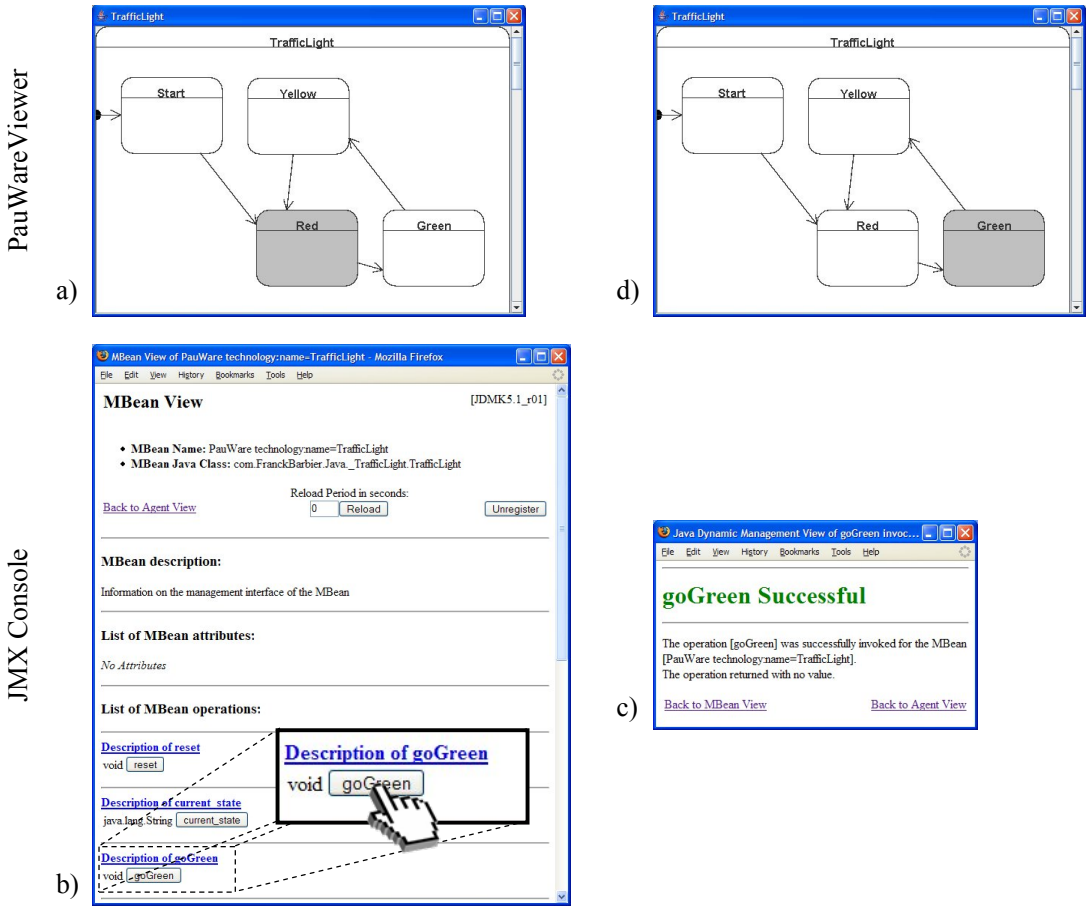


FIG. 5.5 – Administration via une console JMX et surveillance via le PauWareView

réalisée avec succès, ce qui remplit déjà une condition nécessaire à leur administration avec WMX.

Afin de donner un meilleur aperçu des capacités de WMX, nous avons conduit une étude de performance, dont cette section présente l'élaboration et les résultats.

5.4.1 Elaboration du banc d'essai et mesures

Notre étude concerne l'*overhead* produit lors d'un changement d'état. Pour évaluer la surcharge causée par les différents traitements induits par WMX, nous avons suivi une approche inspirée par le banc d'essai utilisé pour évaluer le coût de l'utilisation des aspects dans le *framework* JAsCo [153].

Le principe est de mesurer le temps d'exécution moyen d'itérations de 100000 changements d'état effectués sur des composants de test implémentés selon différentes approches. Tous les composants implémentent un même comportement simple composé de deux états.

Voici les différentes implémentations que nous avons évaluées :

1. **Java simple :** implémentation classique d'un automate sans hiérarchie comme dans [118] qui utilise les constructions de base du langage (*switch*).
2. **Java simple + API *reflect* :** même implémentation que la précédente sauf pour les changements d'état qui sont initiés via les mécanismes de réflexion Java.
3. **JMX avec contrôle interne :** intègre l'implémentation « Java simple » dans un serveur JMX. Les changements d'état sont alors effectués par des contrôles internes au serveur.

4 et 5. PauWare, avec et sans cache : deux implémentations PauWare sont testées, une utilisant un mécanisme de cache des transitions et l'autre sans.

6 et 7. Velcro, avec et sans cache : deux implémentations Velcro sont testées, une utilisant un mécanisme de cache des transitions et l'autre sans.

8. **Java simple avec console de log :** même implémentation que celle de « Java simple » mais avec en plus une console de log (`System.out.println()`) traçant les changements d'état.

9 et 10. WMX avec sockets : deux implémentations WMX sont évaluées, une côté composant WMX et l'autre côté système d'administration. Les deux implémentations utilisent un adaptateur à base de sockets similaire à l'adaptateur basé sur le connecteur réseau générique de Java ME (`javax.microedition.io.Connector`). La première implémentation utilise cet adaptateur pour transmettre les changements d'état et la seconde l'utilise pour envoyer les ordres de contrôle sur le composant distant.

11. **JMX avec adaptateur RMI :** par rapport à la troisième implémentation les changements d'état sont internes mais initiés à distance via un adaptateur RMI.

Les expérimentations que nous avons menées ont été faites sur un Pentium M 1,6 Ghz avec 512 Mo de RAM exécutant la JVM de Java 1.5 sur Windows XP. Les implémentations mettant

en jeu des connexions réseaux ont été réalisées localement pour minimiser l'impact du délai des communications sur les résultats. Nous n'avons pas choisi d'appareil sans fil comme plate-forme de test afin de pouvoir comparer WMX et des implémentations comme JMX qui ne peuvent pas être exécutées sur Java ME. Le tableau 5.1 présente les mesures que nous avons relevées sur notre plate-forme de test.

N°	**Implémentation**	**Temps d'exécution moyen relevé**	**Overhead par changement d'état**
1	Java simple	2 ms	0 μs
2	Java simple + API *reflect*	14 ms	0,12 μs
3	JMX (accès interne)	721 ms	7,19 μs
4	PauWare (avec cache)	1027 ms	10,25 μs
5	PauWare (sans cache)	1491 ms	14,89 μs
6	Velcro (avec cache)	1038 ms	10,36 μs
7	Velcro (sans cache)	1529 ms	15,27 μs
	Les implémentations suivantes mettent en jeu des entrées/sorties ou des connexions réseaux		
8	Java simple + console de log	2584 ms	25,82 μs
9	WMX côté sans fil	3893 ms	38,91 μs
10	WMX côté administration	73 ms	0,71 μs
11	JMX + adaptateur RMI	22077 ms	220,75 μs

TAB. 5.1 – Banc d'essai mesurant le temps d'exécution de 100000 changements d'état sur différentes implémentations

5.4.2 Analyse des résultats

Les sept premières expérimentations n'évaluent pas directement WMX mais l'implémentation des machines à états avec PauWare et Velcro. La première mesure, qui est effectuée sur une implémentation simple dans laquelle les changements d'état sont provoqués par des appels de méthodes, sert d'étalon (0 *overhead*) à notre série de mesures.

Tout d'abord, les résultats nous permettent d'observer que les librairies PauWare et Velcro ont des performances comparables, avec seulement 1% et 2,5% d'*overhead* supplémentaire pour les implémentations Velcro. On aurait pu penser que la librairie Velcro serait même plus performante puisque cette version n'utilise pas la réflection Java qui, comme le montre la deuxième mesure (+0,12 μs), pénalise l'exécution. Néanmoinsn c'est un résultat très satisfaisant puisque le but recherché avec Velcro n'était pas la performance et que des optimisations sont encore possibles comme nous l'avons vu en section 5.1.2.

Les essais effectués sur les différentes implémentations PauWare et Velcro nous permettent également d'apprécier le bénéfice du mécanisme de cache des transitions avec −30% d'*overhead* sur PauWare et −32% sur Velcro. L'ajout dynamique des transitions, que nous avons par exemple

utilisé dans WMX pour la mise à jour des machines à états des gestionnaires externes, est donc à réserver pour des usages spécifiques.

Par rapport aux mécanismes internes de JMX (mesure n°3), il n'est pas étonnant de relever de moins bonnes performances pour les implémentations de PauWare et Velcro (entre +42% et +133% selon les cas) compte tenu des contrôles effectués par les machines à états.

En ce qui concerne véritablement WMX, la mesure n°9 relève l'*overhead* généré par l'observation d'un composant WMX. On observe alors seulement 50% d'*overhead* supplémentaire par rapport à une simple console de log (mesure n°8) qui ne fait qu'afficher les changements d'état, alors que WMX le fait en mettant en jeu des connexions réseaux. Au niveau du contrôle distant des composants, les performances de WMX surpassent largement celles de JMX utilisé avec un adaptateur RMI, avec 0,71 μs d'*overhead* par changement d'état contre 220,75 μs pour JMX. L'analyse est certes faussée par le fait que WMX utilise un mode de traitement asynchrone et n'attend pas le résultat du contrôle au contraire de JMX. Malgré tout, même en additionnant le temps des différentes étapes dans le traitement d'une action de contrôle, c'est-à-dire l'implémentation n°10 pour l'envoi, la n°9 pour le traitement par le gestionnaire interne et la n°4 pour la mise à jour de la machine à états du gestionnaire externe (implémentée avec PauWare en utilisant les transitions dynamiques), on obtient 5457 ms contre 22077 ms. La différence de support de communication, RMI contre sockets, entre certainement encore en jeu. Cependant, à défaut de permettre une véritable comparaison, cela montre néanmoins que l'*overhead* induit par WMX entre dans des durées acceptables pour l'administration puisque JMX est une référence en la matière dans le monde Java.

Conclusion et perspectives

Dans cette dernière partie, nous dressons le bilan de notre travail de thèse et en résumons les principales contributions. Puis, nous présentons des pistes de recherche qu'il serait intéressant de poursuivre dans la continuité de nos travaux.

Bilan et contributions

Dans cette thèse, nous nous sommes intéressés à l'administration des composants logiciels dans les systèmes sans fil. L'objectif était de permettre l'observation et le contrôle à distance du comportement des composants embarqués dans de tels systèmes.

Pour cela, nous avons proposé une approche qui prend en compte l'administration du comportement dès la conception des composants en permettant de développer des composants logiciels administrables. Le principe repose sur une technique d'exécution de modèles : le comportement des composants est décrit par des machines à états UML 2 qui sont directement exécutées. Pour les besoins de l'administration, ce comportement est alors retrouvé au moment de l'exécution via les machines à états.

Nous avons ainsi défini une architecture d'administration qui s'appuie sur des gestionnaires internes pour contrôler les machines à états au sein même des composants logiciels déployés dans les systèmes sans fil. Au niveau du système d'administration, les machines à états des composants sont répliquées dans des gestionnaires externes. Afin de maintenir l'observation de l'état des composants à jour, nous avons conçu un mécanisme de synchronisation des machines à états entre gestionnaires internes et gestionnaires externes. Notre architecture offre ainsi la possibilité d'élever le raisonnement de l'administration grâce à des états logiques qui représentent le comportement des composants tel qu'il a été établi pendant leur conception. De même, l'administration peut s'appuyer sur différents types de contrôle permis par les machines à états tels que le contrôle par événement, le contrôle par état et le contrôle par action. Nous avons ainsi pu expérimenter de nouvelles politiques d'administration basées sur ces considérations de haut niveau d'abstraction.

Notre architecture a été évaluée à travers l'implémentation de WMX. Cette dernière supporte notre approche en permettant à la fois le développement de composants WMX administrables dans les systèmes sans fil et le développement de leurs gestionnaires comportementaux dans le

système d'administration distant. En amont de l'implémentation de WMX, nous avons développé sur la base de la librairie PauWare, une machine à états exécutable, Velcro, adaptée aux systèmes sans fil. En aval, nous facilitons le contrôle et la visualisation du comportement des composants en proposant des consoles et des outils de visualisation graphique.

Enfin, nous avons montré que notre approche était fonctionnelle en administrant via WMX des composants déployés sur des PDA et nous avons étudié de façon plus précise l'applicabilité de notre approche en mesurant la charge supplémentaire induite par notre solution en terme de temps d'exécution. Sans être optimaux, les résultats ont révélé des performances acceptables comparées à certaines configurations utilisées par le standard d'administration JMX.

Perspectives de recherche

Différentes perspectives de recherche peuvent être envisagées pour poursuivre nos travaux : à court terme comme une prolongation de WMX, à moyen terme comme une application dans des domaines connexes, et à plus long terme pour véritablement permettre une administration dirigée par les modèles.

A. Optimisation, extension et intégration de WMX

Lors de l'élaboration de WMX, nous nous sommes concentrés sur la réalisation efficace de l'architecture d'administration que nous avons conçue. Nous n'avons ainsi pas pu nous attarder sur certains aspects qu'il aurait été intéressant d'approfondir et d'implémenter. Nous évoquons ici certains de ces compléments qui peuvent être apportés à notre prototype pour en améliorer les performances, en faciliter l'utilisation et en permettre l'intégration dans un modèle de composants établi.

A.1 Optimisation de la machine à états

Pour réaliser l'implémentation des machines à états Velcro, nous nous sommes appuyés sur la librairie PauWare et en avons étendu le développement pour qu'elle soit applicable sur des systèmes sans fil. Nous avons ainsi conçu un mécanisme qui permet d'éviter l'utilisation de la réflexion Java dans PauWare. Parmi d'autres optimisations possibles dans la librairie, ce mécanisme pourrait servir de base pour une extension de PauWare utilisant un mécanisme plus performant que la réflexion Java connue pour être coûteuse et pénalisante au niveau des performances. Comme nous en avons discuté en 5.1.2, une approche similaire à celle exploitée dans *SmartReflection* pourrait être exploitée [28].

A.2 Extension pour des politiques dynamiques

Au cours de cette thèse, nous avons pu apprécier les similitudes entre les règles ECA utilisées classiquement dans les systèmes d'administration pour définir des politiques et les transitions des machines à états. En section 4.4.2, nous avons justement utilisé les mécanismes des machines à états au niveau de la construction des gestionnaires externes pour définir notre politique d'administration. Il serait intéressant de pouvoir définir dynamiquement de nouvelles politiques d'administration en modifiant directement la machine à états répliquée dans les gestionnaires. Les règles ECA n'apparaîtraient plus sous forme de liste mais seraient alors organisées en fonction des états du comportement des composants.

A.3 WMX sur OSGi

Afin de nous concentrer sur notre problématique de l'administration du comportement des composants, nous avons bâti notre approche sur un modèle de composants simple à base de classes et d'interfaces. A l'instar du travail d'incorporation de PauWare dans les EJB qui a été réalisé dans [10], il serait intéressant pour promouvoir l'utilisation de WMX, d'appliquer notre solution à un modèle de composants comme OSGi qui connaît une popularité croissante dans le monde du sans fil.

B. Domaines d'investigation futurs

De nombreux domaines pourraient être explorés pour poursuivre ce travail de thèse. Nous proposons trois directions pour des travaux de recherche à moyen terme concernant l'*Autonomic Computing*, l'administration de services et l'administration de capteurs dans les réseaux de capteurs.

B.1 Application à l'*Autonomic Computing*

Les travaux de cette thèse ont été pensés comme une étape préliminaire pour effectuer des recherches sur l'*Autonomic Computing*. Au niveau de PauWare, l'utilisation d'invariants d'états pourrait permettre de synchroniser les états des machines à états avec les données métier de leur composant. Le lecteur peut se référer à la section 5.8 de [11] intitulée, *States and business data consistency checking through state invariants*, qui présente des expérimentations embryonnaires sur l'utilisation des invariants dans la conception de mécanismes de rétablissement autonomiques. Ces recherches sont poursuivies dans la thèse de Cyril Ballagny en cours de réalisation au LIUPPA[28] et qui s'intitule « Composants et services logiciels autonomiques, self-managing par intéraction complexe ».

[28]Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour

Notre approche semble particulièrement bien adaptée à la gestion de propriétés de *self-healing* ou de *self-repairing* telles qu'elles sont envisagées par exemple dans le projet SHADOWS (Self-Healing Approach to Designing cOmplex softWare Systems)[29]. Dans cette optique, l'extension de WMX pour des politiques dynamiques serait une première voie d'exploration intéressante afin de lier notre approche à un organe d'analyse et de planification autonomique.

B.2 Application à l'administration de services

L'administration est une activité particulièrement importante dans les systèmes orientés service. Notre approche pourrait être appliquée à ce domaine dans la lignée des travaux de Spanoudakis *et al.* [141] qui permettent d'observer des propriétés comportementales de systèmes orientés service à partir d'un *event calculus*. Ce travail pourrait être initié par l'expérimentation de WMX sur OSGi que nous avons évoquée précédemment.

B.3 Application à l'administration de capteurs

Suivant les gains de performances obtenus par le travail d'optimisation de la machine à états dont nous avons parlé, notre approche pourrait être appliquée à la conception et l'administration de capteurs dans les réseaux de capteurs. Les machines à états sont particulièrement intéressantes pour ce type d'appareils dont le comportement doit permettre la gestion de nombreux événements [82].

C. Vers une administration dirigée par les modèles

Enfin, à plus long terme, nos travaux pourraient s'inscrire dans une démarche globale d'administration dirigée par les modèles (et probablement autonomique) dans laquelle les différentes vues d'un système, telles que modélisées par UML, sont retrouvées à l'exécution sous forme de modèles et exploitées en tant que telles pour élever le niveau d'abstraction global de l'activité d'administration.

[29] `https://sysrun.haifa.il.ibm.com/shadows/`

Bibliographie

[1] Khaldoun Al Agha, Guy Pujolle, and Guillaume Vivier. *Réseaux de mobiles & réseaux sans fil*. Eyrolles, 2001. 24

[2] Armen Aghasaryan, Eric Fabre, Albert Benveniste, Renée Boubour, and Claude Jard. Fault Detection and Diagnosis in Distributed Systems : An Approach by Partially Stochastic Petri Nets. *Discrete Event Dynamic Systems*, 8(2) :203–231, 1998. 61

[3] Nada Almasri and Stéphane Frénot. Dynamic Instrumentation for the Management of EJB-based Applications. Technical Report 4481, INRIA Rhône-Alpes, 06 2002. 52

[4] Laurent Andrey, Olivier Festor, and Abdelkader Lahmadi. Evaluation du passage à l'échelle des systèmes de gestion : métriques et modèles. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP'06)*, Tozeur, Tunisie, 2006. Hermès. 42

[5] The Apache Software Foundation. *Excalibur*. `http://www.apache.org`. 20

[6] Juan I. Asensio, Víctor A. Villagrá, Jorge E. López de Vergara, Roney Pignaton, and Julio J. Berrocal. Experiences in the management of an EJB-based e-commerce application. In *Proceedings of the 8th HP OpenView University Association Plenary workshop (HPOVUA'01)*, 2001. 52

[7] Collins Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Peach, Jurgen Wust, and Jorg Zettel. *Component-Based Product Line Engineering with UML*. Component Software. Addison-Wesley, 2001. 6

[8] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, R.J. Seacord, and K. Wallnau. Volume II : Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000. 6, 11, 19

[9] Franck Barbier. *UML 2 et MDE*. Dunod, 2005. 78

[10] Franck Barbier. An Enhanced Composition Model for Conversational Enterprise JavaBeans. In *9th International SIGSOFT Symposium on Component-Based Software Engineering*, pages 344–351, Mälardalen University,Västerås, Sweden, 2006. 95, 145

[11] Franck Barbier. PauWare Users' Guide, version 1.1. Technical report, PauWare Research Group, décembre 2006. `http://www.pauware.com`. 91, 130, 132, 137, 145

[12] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Softw. Eng.*, 29(5) :459–470, 2003. 119, 121

[13] Franck Barbier and Fabien Romeo. *Handbook of Research on Mobile Business*, chapter Administration of Wireless Software Components, pages 200–215. Idea Group Publishing, 2006. 111

[14] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward Open-World Software : Issue and Challenges. *IEEE Computer*, 39(10) :36–43, 2006. 30

[15] Michael Beisiegel, Henning Blohm, Dave Booz, Jean-Jacques Dubray, Adrian Colyer, Mike Edwards, Don Ferguson, Bill Flood, Mike Greenberg, Dan Kearns, Jim Marino, Jeff Mischkinsky, Martin Nally, Greg Pavlik, Mike Rowley, Ken Tam, and Carl Trieloff. *Service Component Architecture, Building Systems using a Service Oriented Architecture.* BEA, IBM, Interface21, IONA, Oracle, SAP, Siebel, Sybase, white paper, version 0.9 edition, novembre 2005. 10

[16] Nicolas Belloir. *Composition conceptuelle basée sur la relation Tout-Partie.* PhD thesis, Université de Pau et des Pays de l'Adour, 2004. 8

[17] Morgan Björkander and Cris Kobryn. Architecting Systems with UML 2.0. *IEEE Software*, 2003. 8

[18] Bluetooth SIG. *Specification of the Bluetooth System*, 2004. 26

[19] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998. 77

[20] Raouf Boutaba and Jin Xiao. Network Management : State of the Art. In *Proceedings of the IFIP 17th World Computer Congress - TC6 Stream on Communication Systems : The State of the Art*, pages 127–146, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. 40

[21] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2) :323–342, 1983. 60

[22] Jean-Michel Bruel, Franck Barbier, Nicolas Belloir, and Fabien Romeo. *Ingénierie des composants : concepts, techniques et outils*, chapter Test de Composants Logiciels, pages 229–245. Vuibert, Paris, 2005. 2

[23] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java : Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12) :1257–1284, 2006. 18

[24] Luiz Eduardo Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* PhD thesis, University of Newcastle upon Tyne, 1994. 102, 103

[25] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). Technical Report RFC 1157, The Internet Society, mai 1990. 36, 42

[26] J. Case, M. Fedor, M. Schoffstall, and J. Davin. Manager-to-Manager Management Information Base. Technical Report RFC 1451, The Internet Society, Avril 1993. 43

[27] Corine Cauvet and Philippe Ramadour. *Ingénierie des composants : concepts, techniques et outils*, chapter Les composants métiers dans l'ingénierie des systèmes d'information, pages 197–228. Vuibert, Paris, 2005. 19

[28] Walter Cazzola. SmartReflection : Efficient Introspection in Java. *Journal of Object Technology*, 3(11) :117–132, December 2004. Special issue : OOPS track at SAC 2004, Nicosia, Chyprus. 128, 144

[29] CCITT (ITU-T), Genève, Suisse. *Recommendation M.3400. TMN Management Functions*, Octobre 1992. 37

[30] CCITT (ITU-T), Genève, Suisse. *Recommendation X.700. Data Communication Network – Management Framework for Open Systems Interconnection (OSI) for CCITT Applications*, Septembre 1992. 36

[31] CCITT (ITU-T), Genève, Suisse. *Recommendation X.701. Data Communication Networks – Information Technology – Open Systems Interconnection – Systems Management Overview*, Janvier 1992. 35

[32] CCITT (ITU-T), Genève, Suisse. *Recommendation X.722. Data Communication Networks – Information Technology – Open Systems Interconnection – Structure of Information Management : Guidelines for the Definition of Managed Objects*, Janvier 1992. 36

[33] CCITT (ITU-T), Genève, Suisse. *Recommendation M.3010. Principles for a Telecommunications Management Network*, Février 2000. 43

[34] CCITT (ITU-T), Genève, Suisse. *Recommendation X.681. Information Technology – Abstract Syntax Notation One (ASN.1) : Information Object Specification*, Juillet 2002. 36

[35] S. Chakravarthy, B. Blaustein, A. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC : A Research Project in Active, Time-Constrained Database Management. Technical report, Final Technical Report, XAIT-89-02, XAIT Reference Number 187, Xerox Advanced Information Technology, July 1989. 55, 58, 83

[36] T. Coupaye, V. Quéma, L. Seinturier, and J.-B. Stefani. *Intergiciel et Construction d'Applications Réparties*, chapter Le modèle de composants Fractal, pages 49–76. Creative Commons, Grenoble, janvier 2007. 18

[37] Thierry Coupaye and Nicolas Rivierre. Observing Component Behaviors with TLA. In *Workshop on Correctness of Model-based Software Composition (ECOOP 2003)*, Darmstadt, Germany, 2003. 59, 60

[38] Ivica Crnkovic. Component-based Software Engineering for Embedded Systems. In *International Conference on Software engineering, ICSE'05*, St. Luis, USA, May 2005. ACM. 30

[39] Ivica Crnkovic, Brahim Hnich, Torsten Jonsson, and Zeynep Kiziltan. *Building Reliable Component-Based Software Systems*, chapter Basic Concepts in CBSE, pages 3–22. Artech House Publishers, Boston, 2002. 6

[40] Pierre-Charles David. *Dévoloppement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation.* PhD thesis, Institut National Polytechnique de Grenoble, 2006. 57

[41] Distributed Management Task Force. *Common Information Model (CIM) Specification Version 2.2*, Juin 1999. DSP 0004. 36

[42] Distributed Management Task Force. *Web-Based Enterprise Management (WBEM)*, 2007. `http://www.dmtf.org/standards/wbem`. 37

[43] Klaus R. Dittrich, Stella Gatziu, and Andreas Geppert. The active database management system manifesto : A rulebase of adbms features. In *RIDS '95 : Proceedings of the Second International Workshop on Rules in Database Systems*, pages 3–20, London, UK, 1995. Springer-Verlag. 55, 58, 83

[44] Didier Donsez. *Intergiciel et Construction d'Applications Réparties*, chapter La plate-forme dynamique de services OSGi, pages 195–224. Creative Commons, Grenoble, janvier 2007. 28

[45] Bruce Powel Douglass. *Real-Time UML : developing efficient objects for embedded systems.* Addison-Wesley, second edition, 2000. 74

[46] Desmond Francis D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML : The Catalysis Approach.* Addison-Wesley, 1998. 9

[47] A. Farrel and H. Kreger. Web Services Management Approaches. *IBM Systems Journal*, 41(2) :212–227, 2002. 38

[48] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles. Au-delà du MDA.* Hermes, 2006. 87

[49] Olivier Festor. Ingénierie de la gestion de réseaux et de services : du modèle OSI à la technologie active. Technical report, Loria, décembre 2001. HDR. 34, 35

[50] Olivier Festor and Laurent Andrey. *Standards pour la gestion des réseaux et des services*, chapter JMX : un standard pour la gestion Java, pages 217–254. Hermès Science Publishing, 2004. 51

[51] Olivier Festor and Radu State. Supervision et contrôle des réseaux et services : retour vers le futur. In *Cinquième Ecole d'Hiver des Télécommunications - ECOTEL'2002*, Golfe Juan, France, 2002. 40

[52] S. Frénot and D. Stefan. Instrumentation de plates-formes de services ouvertes - Gestion JMX sur OSGi. In *1eres journées francophones Mobilité et Ubiquité*, pages 199–202, Nice, France, 2004. ACM. 54

[53] Geoffrey Frost. Motorola's Edge. *The Hub Magazine*, septembre/octobre 2005. 26

[54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, January 1995. 12

[55] A.G. Ganek and T.A. Corbi. The Dawning of the Autonomic Computing Era. *IBM Systems Journal*, 42(1) :5–18, 2003. 44

[56] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch or Why it's hard to build systems out of existing parts. In *17th International Conference on Software Enginneering*, pages 179–185, Seattle, Washington, 1995. ACM SIGSOFT. 119

[57] D. Garlan, R. Monroe, and D. Wile. Acme : An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, Toronto, Canada, novembre 1997. 11

[58] Damianos Gavalas, Dominic Greenwood, Mohammed Ghanbari, and Mike O'Mahony. Hierarchical network management : a scalable and dynamic mobile agent-based approach. *Comput. Networks*, 38(6) :693–711, 2002. 42

[59] Wolfgang Grieskamp, Maritta Heisel, and Heiko Dörr. Specifying Embedded Systems with Statecharts and Z : An Agenda for Cyclic Software Components. *Lecture Notes in Computer Science*, 1382 :88–115, 1998. 74

[60] I. Grosclaude. Model-based Monitoring of Component-Based Software Systems. In *16th European Conference on Artificial Intelligence*, pages 1025–1026, Valencia, Spain, 2004. IOS Press. 61

[61] R. S. Hall, D. Heimbigner, and A. L. Wolf. Software Deployment Languages and Schema. Technical report, Technical Report CUSERL-203-97, University of Colorado Software Engineering Research Laboratory, décembre 1997. 8

[62] David Harel. Statecharts : A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3) :231–274, June 1987. 74, 82, 83, 85

[63] D. Heimbigner, A. Wolf, R. Hall, and A. van der Hoek. Deploying Component Software Systems. In *Workshop on Compositional Software Architectures*, Monterey, California, USA, Janvier 1998. 8

[64] Brian Henderson-Sellers, Rajesh Pradhan, Clemens Szyperski, Antero Taivalsaari, and Alan Wills. Are Components Objects ? In *Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'99 Companion, Panel Discussion*, Denver, Colorado, November 1999. 9

[65] Hans Hrasna. *Java 2 Platform, Enterprise Edition Management Specification, JSR-77, Maintenance Release v1.1.* Sun Microsystems, Inc., mai 2006. 51

[66] Michael N. Huhns and Munindar P. Singh. Services-Oriented Computing : Key Concepts and Principles. *IEEE Internet Computing*, 9(1) :75–81, 2005. 10

[67] IBM. *An architectural blueprint for autonomic computing*, white paper, third edition edition, juin 2005. 45

[68] IEEE. *ANSI/IEEE Std 802.11, 1999 Edition*, 1999. 26

[69] IEEE. *IEEE Std 802.16*, 2003. 26

[70] ITU-T, Genève, Suisse. *Recommendation X.711. Information Technology – Open Systems Interconnection – Common Management Information Protocol : Specification*, Octobre 1997. 36

[71] Harris J. and Henderson A. A Better Mythology for System Design. In *Computer Science-Human Computer Interaction Conference*, Pittsburgh, USA, mai 1999. 6

[72] Rod Johnson. *Expert One-on-One J2EE Design and Development.* Wrox, 2002. 16

[73] Rod Johnson and Juergen Hoeller. *Expert One-on-One J2EE Development without EJB.* Wrox, 2004. 20

[74] Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Colin Sampaleanu. *Professional Java Development with the Spring Framework.* Wrox, 2005. 16

[75] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4) :596–619, 1983. 60

[76] M. Kahani and H. W. P. Beadle. Decentralized Approaches for Network Management. *Computer Communications Review*, 27(3), 1997. 40

[77] Stuart Kent. Model Driven Engineering. In *3rd International Conference on Integrated Formal Method*, pages 286–298. Springer, 2002. 87

[78] Jeffrey Kephart and David Chess. The Vision of Autonomic Computing. *Computer Magazine*, 36(1) :41–50, 2003. 44, 45

[79] Abdelmadjid Ketfi. *Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels.* PhD thesis, Université Joseph Fourier, 2004. 54

[80] Gregor Kiczales. Beyond the Black Box : Open Implementation. *IEEE Software*, 13(1) :8–11, 1996. 108

[81] Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *ICSE '97 : Proceedings of the 19th international conference on Software engineering*, pages 481–490, New York, NY, USA, 1997. ACM Press. 108

[82] Tae-Hyung Kim and Seongsoo Hong. State Machine Based Operating System Architecture for Wireless Sensor Networks. In *5th International Conference on Parallel and Distributed Computing : Applications and Technologies*, pages 803–806, 2004. 146

[83] Charles D. Knutson and Jeffrey M. Brown. *IrDA Principles and Protocols : The IrDA Library, Vol. 1.* MCL Press, 2004. 26

[84] H. Kopetz and N. Suri. Compositional design of RT systems : A conceptual basis for specification of linking interfaces. In *6th IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC)*, Hokkaido, Japan, may 2003. 108

[85] H. Kreger. Java Management Extensions for Application Management. *IBM Systems Journal*, 40(1) :104–129, 2001. 38

[86] H. Kreger, W. Harold, and L. Williamson. *Java and JMX.* Addison Wesley, 2003. 35, 38, 39, 47, 52

[87] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages Systems*, 16(3) :872–923, 1994. 59

[88] K. K. Lau, P. V. Elizondo, and Z. Wang. Exogenous Connectors for Software Components. In *Eighth International SIGSOFT Symposium on Component-based Software Engineering.* Springer Verlag, january 2005. 108

[89] Stanisław Leśniewski. *O podstawach matematyki.* Przeglad Filozoficzny, 1927-1931. 119

[90] Martin Mallick. *Mobile and Wireless Design Essentials.* John Wiley & Sons, Ltd., 2003. 27

[91] Robert Cecil Martin. *Agile Software Development : Principles, Patterns and Practices.* Pearson Education, 2002. 16, 20

[92] Jean-Philippe Martin-Flatin. Push vs. Pull in Web-Based Network Management. In *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management (IM'99)*, pages 3–18, Boston, MA, May 1999. 110

[93] Jean-Philippe Martin-Flatin. *Web-based Management of IP Networks and Systems.* PhD thesis, Ecole polytechnique fédérale de Lausanne, 2000. 34

[94] Jean-Philippe Martin-Flatin, Simon Znaty, and Jean-Pierre Hubaux. A Survey of Distributed Enterprise Network and Systems Management Paradigms. *Journal of Network and Systems Management*, 7(1) :9–26, 1999. 40

[95] Peter Marwedel. *Embedded System Design.* Springer, December 2005. 74

[96] M. C. Maston. Using the World Wide Web and Java for Network Service Management. In *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, volume 86, pages 71–84, San Diego, CA, USA, 1997. Chapman & Hall. 42

[97] K. McCloghrie, M. Fine, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, and F. Reichmeyer. Structure of Policy Provisioning Information (SPPI). Technical Report RFC 3159, The Internet Society, août 2001. 36

[98] K. McCloghrie, D. Perkins, and J. Schoenwaelder. Structure of Management Information Version 2 (SMIv2). Technical Report STD 58, RFC 2578, The Internet Society, avril 1999. 36

[99] Douglas McIlroy. Mass-produced Software Components. In J.M. Buxton et al., editor, *Software Engineering Concepts and Techniques, 1968 NATO Conference on Software Engineering*, Garmisch Pattenkirschen, Germany, octobre 1968. 5

[100] N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26, 2000. 11

[101] Stephen J. Mellor and Marc J. Balcer. *Executable UML : a foundation for model-driven architecture.* Addison-Wesley, 2002. 88

[102] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled : Principles of Model-Driven Architecture.* Addison-Wesley, 2004. 88

[103] Bertrand Meyer. On to components. Technical report, IEEE Computer, janvier 1999. 6, 9

[104] Bertrand Meyer. The Significance of Components. *Software Development*, 7(11) :56–57, novembre 1999. 9

[105] Granville Miller, Scott Ambler, Steve Cook, Stephen Mellor, Karl Frank, and Jon Kern. Model driven architecture : the realities, a year later. In *OOPSLA '04 : Companion*

to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 138–140, New York, NY, USA, 2004. ACM Press. `http://blogs.msdn.com/stevecook/archive/2004/10/27/248322.aspx`. 87

[106] Anders Möller, Joakim Fröberg, and Mikael Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *International Symposium on Component-based Software Engineering (CBSE7)*, Edinburgh, Scotland, May 2004. Springer Verlag. 30

[107] Motorola Inc. and Nokia Corporation. *Mobile Operational Management For Java 2 Platform, Micro Edition, JSR-232 Version 1.O Final*, 2006. 67

[108] Pierre-Alain Muller. De la modélisation objet des logiciels à la metamodélisation des langages informatiques. Technical report, Université de Rennes 1, novembre 2006. HDR. 88

[109] Oscar Nierstrasz and Theo Dirk Meijler. Research Directions in Software Composition. *ACM Computing Surveys*, 27(2) :262–264, 1995. 11

[110] Object Management Group. *Model Driven Architecture*. `http://www.omg.org/mda`. 87

[111] Object Management Group. *MOF 2.0 / XMI Mapping Specification, v2.1*, September 2005. 131

[112] Object Management Group. *Unified Modeling Language : Infrastructure, version 2.1.1*, 2007. 80

[113] Object Management Group. *Unified Modeling Language : Superstructure, version 2.1.1*, 2007. 77, 81, 82, 86

[114] Open Mobile Alliance. *Device Management Protocol*, January 2005. 62

[115] Open Mobile Alliance. *DM DiagMon Architecture*, May 2007. 66

[116] Open Mobile Alliance. *Firmware Update Management Object Architecture*, February 2007. 63

[117] Open Mobile Alliance. *Software Component Management Object*, May 2007. 64

[118] C. Enrique Ortiz. Managing the MIDlet Life-Cycle with a Finite State Machine. Technical report, Sun Microsystems, Inc., août 2004. `http://developers.sun.com/mobility/midp/articles/fsm/`. 139

[119] OSGi Alliance. *OSGi Service Platform Mobile Specification, Release 4, Version 4.0*, July 2006. 28, 67

[120] David Parnas. On the Criteria To Be Used in Decomposing Systems Into Modules. *Communications of the ACM*, 15(12) :1053–1058, décembre 1972. 6, 104

[121] Luca Pazzi. Part-Whole Statecharts for the Explicit Representation of Compound Behaviors. In *UML*, pages 541–555, 2000. 119

[122] Frederic Peschanski and Jean-Pierre Briot. *Ingénierie des composants : concepts, techniques et outils*, chapter Architecture de composants répartis, pages 247–280. Vuibert, Paris, 2005. 20, 21

[123] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, 1962. 61

[124] PicoContainer Organization. *PicoContainer*. `http://www.picocontainer.org`. 20

[125] Guy Pujolle. *Les réseaux*, chapter La gestion et le contrôle par politique, pages 833–850. Eyrolles, Paris, 2004. 43

[126] Roger Riggs, Antero Taivalsaari, Jim Van Peursem, Jyri Huopaniemi, Mark Patel, and Aleksi Uotila. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*. Addison-Wesley, second edition edition, 2003. 25

[127] Nicolas Rivierre. *Fractal JMX Tutorial*. France Telecom R & D, 2004. `http://fractal.objectweb.org/tutorials/jmx/index.html`. 54

[128] Nicolas Rivierre, Francois Horn, and Frederic Dang Tran. On Monitoring Concurrent Systems with TLA : An Example. In *ACSD '05 : Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 36–45, Washington, DC, USA, 2005. IEEE Computer Society. 59

[129] Fabien Romeo, Cyril Ballagny, and Franck Barbier. PauWare : un modèle de composant basé état. In *Actes des Journées Composants (JC2006)*, pages 1–10, Canet en Roussillon, France, October 2006. 99

[130] Fabien Romeo and Franck Barbier. Administration de composants logiciels : application aux systèmes sans fil. *Génie Logiciel*, (73) :39–43, juin 2005. 111

[131] Fabien Romeo and Franck Barbier. Management of Wireless Software Components. In *the 10th International Workshop on Component-Oriented Programming in the 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005. 111

[132] Fabien Romeo, Franck Barbier, and Jean-Michel Bruel. Autonomic Management of Component-Based Embedded Software. In *Proceedings of the tenth IFIP/IEEE International Symposium on Integrated Network Management (IM 2007)*, pages 21–25, Munich, Germany, May 2007. 114

[133] Fabien Romeo, Franck Barbier, and Jean-Michel Bruel. Observability and Controllability of Wireless Software Components. In *Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 48–61, Paphos, Chyprus, June 2007. 114

[134] M. Román and N. Islam. Dynamically Programmable and Reconfigurable Middleware Services. In *5th ACM/IFIP/ USENIX International Conference on Middleware*, pages 372–396, Toronto, Canada, 2004. Springer-Verlag. 68, 69, 70, 71

[135] Miro Samek. *Practical Statecharts in C/C++*. CMP Books, 2002. 89

[136] Jürgen Schönwälder. Traditional Approches to Distributed Management. In *19th meeting of the Network Management Research Group(NMRG) of the Internet Research Task Force (IRTF)*, janvier 2006. 41

[137] Bran V. Selic. On the Semantic Foundations of Standard UML 2.0. In *International School on Formal Methods for the Design of Real-Time Systems*, pages 181–199, Bertinoro, Italy, 2004. 95

[138] Sally Shlaer and Stephen J. Mellor. *Object Lifecycles : Modeling the World in States*. Prentice Hall PTR, 1992. 88

[139] Manfred R. Siegl and Georg Trausmuth. Hierarchical network management : a concept and its prototype in (SNMPv2). *Computer Networks and ISDN Systems*, 28(4) :441–452, 1996. 42

[140] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, février 1982. 54

[141] George Spanoudakis and Khaled Mahbub. Non Intrusive Monitoring of Service Based Systems. *International Journal of Cooperative Information Systems*, 15(3) :325–358, 2006. 146

[142] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland. A Concise Introduction to Autonomic Computing. *Journal of Advanced Engineering Informatics, Engineering Applications of Artificial Intelligence, Special Issue on Autonomic Computing and Automation*, 19(3) :181–187, 2005. 40

[143] Rick Sturm and Winston Bumpus. *Foundations of Application Management*. Wiley, 1999. 34, 37, 38

[144] Sun Microsystems, Inc. *JavaBeans Specification, v1.01*, 1997. 12

[145] Sun Microsystems, Inc. *Java Management Extensions Instrumentation and Agent Specification, v1.2*, 2002. 46, 47, 48

[146] Sun Microsystems, Inc. *Connected Limited Device Configuration (CLDC) Specification Version 1.1*, 2003. 24, 27

[147] Sun Microsystems, Inc. *Enterprise JavaBeans Specification, Version 2.1*, novembre 2003. 13, 50

[148] Sun Microsystems, Inc. *JSR-233, J2EE Mobile Device Management and Monitoring Specification, Draft Version 0.5*, September 2005. 68

[149] Sun Microsystems, Inc. *Connected Device Configuration (CDC) Version 1.1.2*, 2006. 24, 27

[150] Sun Microsystems, Inc. *JSR 220 : Enterprise JavaBeans, Version 3.0, EJB Core Contracts and Requirements*, mai 2006. 13, 50

[151] Sun Microsystems, Inc. *JSR 220 : Enterprise JavaBeans, Version 3.0, Java Persistance API*, mai 2006. 15

[152] Daniel Sundmark, Anders Möller, and Mikael Nolin. Monitored Software Components - A Novel Software Engineering Approach. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, Busan, Korea, 2004. IEEE. 30, 59

[153] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco : an aspect-oriented approach tailored for component based software development. In *International Conference on Aspect-Oriented Software Development AOSD*, pages 21–29, Boston, USA, 2003. ACM Press. 139

[154] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley, second edition edition, 2002. 6, 9, 10, 27, 46

[155] Dan Tapscott. *Growing Up Digital : The Rise of the Net Generation.* McGraw-Hill Companies, 1999. 1

[156] Mark Vigder. *The Evolution, Maintenance, and Management of Component-Based Systems*, chapter Component-Based Software Engineering : Putting the Pieces Together, pages 527–539. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. 46

[157] Jeffrey M. Voas. Maintaining Component-Based Systems. *IEEE Software*, 15(4) :22–27, 1998. 31, 104

[158] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). Technical Report RFC 2251, The Internet Society, décembre 1997. 44

[159] Yaron Weinsberg and Israel Ben-Shaul. A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 374–384, Orlando, Florida, USA, mai 2002. ACM. 29

[160] Scott Weiss. *Handheld Usability.* John Wiley & Sons, Ltd., 2002. 23

[161] Andrea Westerinen and Winston Bumpus. The Continuing Evolution of Distributed Systems Management. *IEICE Transactions on Information and Systems*, E86-D(11) :2256–2261, Novembre 2003. 35, 37

[162] James White and David Hemphill. *Java 2 Micro Edition (Java in Small Things).* Manning, 2002. 24

[163] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control . Technical Report RFC 2753, The Internet Society, janvier 2000. 43

[164] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. Technical Report RFC 1777, The Internet Society, mars 1995. 44

[165] Edward Yourdon and Larry Constantine. *Structured design : Fundamentals of a Discipline of Computer Program and Systems Design.* Prentice-Hall, 1979. 6, 7

[166] Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich. The SATIN Component System - A Metamodel for Engineering Adaptable Mobile Systems. *IEEE Transactions on Software Engineering*, 32(11) :910–927, octobre 2006. 29

[167] Mohammad Zulkernine and Rudolph Seviora. Towards automatic monitoring of component-based software systems. *Journal of Systems and Software*, 74(1) :15–24, 2005. 60, 61

Résumé

Une des grandes tendances dans les systèmes sans fil est de concevoir leur logiciel comme un assemblage de composants. Les composants sont interconnectés grâce à leurs interfaces, tout en cachant leur implémentation afin d'augmenter leur réutilisabilité et permettre leur déploiement par des tiers. Le déploiement s'effectue sur des dispositifs très variés tels que des téléphones mobiles, des PDA, des boîtiers de télévision numérique, etc. Compte tenu du fait que les environnements de déploiement sont différents des environnements de développement, des comportements anormaux et/ou des dysfonctionnements se produisent, ce qui, par conséquent, met en évidence le besoin crucial d'un système d'administration.

Pour répondre à cette problématique, nous avons conçu une architecture pour l'observation et le contrôle à distance du comportement des composants logiciels sans fil. Le comportement est décrit par des machines à états UML 2 qui sont directement exécutées par les composants et répliquées sur le système d'administration. Des politiques d'administration basées sur les états des composants peuvent alors être définies et appliquées. Notre proposition a été évaluée et mise en œuvre dans la technologie WMX (*Wireless Management eXtensions*), qui permet, à la fois, le développement dans les systèmes sans fil de composants logiciels administrables et la construction de gestionnaires basés sur le comportement dans le système d'administration.

Mots-clés: administration, composants logiciels, systèmes sans fil, machine à états.

Abstract

A major trend in wireless systems is the design of their software as an assembly of components. Components are interconnected through their interfaces, while hiding their implementations in order to increase their reuse and to allow to be deployed by third parties. Deployment occurs on various and sundry devices such as mobile phones, PDAs, set-top boxes, and so on. Owing to the fact that deployment environments are different from development environments, abnormal behaviors and/or misuses occur and, consequently, call for remote administration.

In this perspective, we designed an architecture for monitoring and controlling the behavior of wireless software components. This behavior is described by UML 2 state machines, which are directly executed on the wireless system and replicated on the administration system. Management policies based on the components' states can thus be defined and enforced. Our proposal has been evaluated and implemented in the WMX (*Wireless Management eXtensions*) technology, which provides the necessary facilities to build both the manageable software components in a wireless system and their behavioral managers in an administration system.

Keywords: management, software components, wireless systems, state machines.