HAL open science

Sam : un environnement d'exécution pour les applications à services dynamiques et hétérogènes

Eric Simon

To cite this version:

Eric Simon. Sam : un environnement d'exécution pour les applications à services dynamiques et hétérogènes. Mathématiques générales [math.GM]. Université de Grenoble, 2011. Français. NNT : 2011GRENM008 . tel-00585623

HAL Id: tel-00585623

https://theses.hal.science/tel-00585623v1

Submitted on 13 Apr 2011

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Eric SIMON

Thèse dirigée par **Jacky ESTUBLIER**

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

SAM : un environnement d'exécution pour les applications à services dynamiques et hétérogènes.

Thèse soutenue publiquement le **07 mars 2011**,
devant le jury composé de :

Mme. Isabelle DEMEURE Professeur à Télécom ParisTech,	**Rapporteur**
M. Pierre-Alain MULLER Professeur à l'Université de Haute Alsace, Mulhouse,	**Membre**
M. Mourad Chabane OUSSALAH Professeur à l'Université de Nantes,	**Membre**
Mme. Brigitte PLATEAU Professeur à l'ENSIMAG et Directeur de l'ENSIMAG,	**Présidente**
M. Lionel SEINTURIER Professeur à l'Université de Lille 1,	**Rapporteur**
M. Jacky ESTUBLIER Directeur de recherche au CNRS (Grenoble),	**Membre**

Remerciements

Je voudrais remercier tous les membres de mon jury. Je remercie Isabelle DEMEURE et Lionel SEINTURIER pour avoir accepté de rapporter mes travaux de thèse. Je remercie également Pierre-Alain MULLER, Mourad Chabane OUSSALAH et Brigitte PLATEAU pour avoir examiné mes travaux.

Je tiens à remercier mon directeur de thèse Jacky ESTUBLIER pour ses conseils et son aide durant cette thèse. Je souhaite aussi remercier M. Pierre-Yves CUNIN et M. Philippe LALANDA pour m'avoir accueilli dans l'équipe ADELE. De manière plus générale, je remercie l'ensemble des membres de l'équipe ADELE pour l'ambiance aussi bien au travail qu'à l'extérieur. Je tiens particulièrement à remercier Stéphanie et Vincent pour les « conseils » sur la pédagogie ainsi que German pour les nombreuses discussions enrichissantes aussi bien techniques que conceptuelles. Je salue Yoann avec qui j'ai partagé ces trois années de thèses ponctuées de nombreuses discussions techniques, conceptuelles et de beaucoup d'autres sujets qui n'ont absolument rien à voir avec l'informatique.

Je remercie mes parents : Jean-Pierre et Martine ainsi que mon frère : Ehouarn, pour leur soutien et leurs conseils. Je souhaite remercier également l'ensemble de mes amis Nantais et Grenoblois.

Et finalement, mes sentiments les plus chaleureux sont pour mon épouse : Elodie qui m'a soutenu, supporté et réconforté ces 8 dernières années.

Résumé

Ces dernières années, le contexte d'exécution des applications a beaucoup évolué. Nous sommes passés, en moins de 15 ans, d'applications monolithiques et statiques faiblement distribuées à des applications réparties à grande échelle ayant un comportement très dynamique. Les plates-formes d'exécution qui étaient autrefois fermées sont désormais ouvertes à des équipements nomades et aux contextes des utilisateurs comme dans l'informatique ubiquitaire ou la domotique. Les éléments logiciels, dans ce nouveau contexte, peuvent apparaitre ou disparaître de façon imprédictible (ils sont dits « dynamiques ») et sont souvent issus de technologies hétérogènes (Web Service, OSGi, UPnP, etc.). Le caractère imprévisible de l'environnement, et le fait qu'il faille s'y adapter rend, apparemment, les plates-formes d'exécution non déterministes. Il est impératif de pouvoir maîtriser l'évolution des applications dynamiques.

Cette thèse définit une plate-forme pour l'exécution répartie de services hétérogènes. Cette plate-forme fournit un modèle homogène de l'état des services à l'exécution et des opérations d'administration du cycle de vie d'un service, indépendamment des plates-formes réelles. Cette plate-forme fournit également des mécanismes qui permettent d'étendre la supervision et l'administration à d'autres préoccupations (déploiement, versionnement, etc.). Les propriétés d'introspection et de réflexion ainsi obtenues permettent de contrôler l'évolution à l'exécution de l'architecture d'une application et par là même de maîtriser les applications dynamiques. Cette thèse fournit un prototype d'une telle plate-forme nommée SAM-RT.

Mots-Clés : Approche Orientée Service, Composant Orienté Service, ingénierie dirigée par les modèles, model@runtime, adaptabilité

Abstract

During recent years, the execution context of modern applications has evolved. We have moved, in less than 15 years, from monolithic and static applications that were weakly distributed to large-scale distributed applications with very dynamic behavior. The execution platforms, which were otherwise closed, are now open to mobile devices and to user's contexts as is the case with ubiquitous computing and home applications. Software units in this new context can appear and disappear very unpredictably (this is called dynamic behavior) and are often a series of heterogeneous technologies (e.g., Web Services, OSGi, UPnP). The unpredictability of the environment, and the fact that we must adapt to it, makes the execution platforms seem non-deterministic. It is imperative to control the evolution of dynamic applications.

This dissertation defines an execution platform for the distributed execution of heterogeneous services. This platform provides a homogeneous model of the execution state of services and of the administration operations related to the lifecycle of a service, regardless of the underlying technologies. This platform also provides mechanisms to extend monitoring and administration to other concerns (e.g., deployment, versioning). The introspection and reflection properties provided are used to control the evolution of the application's architecture and thereby control dynamic behavior. This dissertation provides a prototype of such a platform named SAM-RT.

Keywords: Service-Oriented Computing, Service-Oriented Component, Model Driven Engineering, model@runtime, adpatability

Sommaire

CHAPITRE 1 - INTRODUCTION 15

CHAPITRE 2 - ÉTAT DE L'ART 21

1. GENIE LOGICIEL 21

1.1 DEVELOPPEMENT 21
1.2 DEPLOIEMENT 23
1.3 EXECUTION 23
1.4 CREATIVITE, CONNAISSANCE ET CHOIX 24
1.5 SYNTHESE 25

2. DYNAMISME ET ADAPTABILITE 27

2.1 ADAPTABILITE ET DISPONIBILITE : DYNAMIQUE VS. STATIQUE 27
2.1.1 Disponibilité 27
2.1.2 Adaptabilité 29
2.1.3 Dynamisme : mécanisme non-transparent 31
2.2 INFORMATIQUE AUTONOMIQUE 32
2.2.1 Motivation 32
2.2.2 Enjeux de l'informatique autonomique 32
2.2.3 Degré d'Autonomicité 34
2.3 ADAPTABILITE : TECHNIQUES D'IMPLANTATION 34
2.4 SYNTHESE 35

3. MODELES DANS LE CYCLE DE VIE D'UNE APPLICATION 37

3.1 CONCEPT DE BASE 37
3.2 ABSTRACTION ET PREOCCUPATIONS 38
3.2.1 Abstraction : une notion de Granularité 38
3.2.2 Séparation des préoccupations 39
3.3 FORMALISME 39
3.4 MODELE A L'EXECUTION 43
3.5 SYNTHESE 46

4. PARADIGMES 47

4.1 APPROCHE A COMPOSANT 47
4.1.1 Composant: Classe, Instance ou unité de déploiement ? 47
4.1.2 Les applications dans l'approche à composant : Assemblages 49
4.1.3 Extensibilité du contexte des composants : Conteneur 51
4.1.4 Environnement modulaire, mais statique 52
4.1.5 Technologies à Composants 52
4.1.6 Synthèse 53
4.2 APPROCHE ORIENTEE SERVICE 55
4.2.1 Définition 55
4.2.2 Style architectural 56
4.2.3 Service : de l'approche à la technologie 58
4.2.4 Dynamisme et Sélection 59
4.2.5 Synthèse 59
4.3 COMPOSANT A SERVICE 60
4.3.1 Web Services 60
4.3.2 Réalisation d'un Service par un composant 61
4.3.3 Réalisation d'une application par Composition 62

4.3.4 Synthèse ... 63
4.4 COMPOSANT ORIENTE SERVICE ... 65
4.4.1 Concepts et mécanismes ... 65
4.4.2 Logique de composition d'application ... 66
4.4.3 Systèmes représentatifs ... 67
4.4.4 Comparaison ... 69
4.4.5 Synthèse ... 69

5. SYSTEMES ADAPTATIFS ... 71

CHAPITRE 3 - CONTRIBUTION ... 75

1. UNE APPROCHE GLOBALE ... 77

1.1 L'APPROCHE SAM ... 77
1.2 LES PHASES DU CYCLE DE VIE ... 77
1.3 ARCHITECTURE DE L'ENVIRONNEMENT SAM ... 79
1.4 OBJECTIF DE LA THESE ... 80

2. META-MODELE DE SERVICE DANS SAM ... 83

2.1 GROUPE D'EQUIVALENCE ... 83
2.2 COMPOSANT A SERVICE VS. COMPOSANT ORIENTE SERVICE ... 84
2.3 META-MODELE « CŒUR » DE L'APPROCHE SAM ... 85
2.3.1 Spécification ... 85
2.3.2 Implémentation ... 86
2.3.3 Instance ... 87
2.3.4 Identification ... 88
2.4 SYNTHESE : LE META-MODELE SAM CORE ... 90

3. ENVIRONNEMENT D'EXECUTION ... 91

3.1 OPERATIONS ET CARACTERISATIONS ... 91
3.1.1 Spécification ... 93
3.1.2 Implémentation ... 94
3.1.3 Instance ... 95
3.1.4 Dépendance ... 96
3.1.5 Méta-modèle SAM CORE à l'exécution ... 98
3.2 SOA SAM ... 99
3.3 SYNTHESE ... 100

4. ENVIRONNEMENT DISTRIBUE HOMOGENE ... 103

4.1 ENVIRONNEMENT HOMOGENE ... 104
4.2 ABSTRACT MACHINE ... 105
4.2.1 Distribution ... 106
4.2.2 Plate-forme à la carte ... 108
4.2.3 Spécialisation SAM ... 109
4.3 SYNTHESE : ARCHITECTURE DE L'*ABSTRACT MACHINE* ... 111

5. ABSTRACTION DES TECHNOLOGIES : INTEGRATION VERTICALE ... 113

5.1 ARCHITECTURE D'INTEGRATION ... 113
5.1.1 Technologies à service ... 113
5.1.2 SAM-CORE-RT générique ... 114
5.1.3 SAM-CORE-RT spécifique ... 114
5.1.4 Intégration de la SAM dans une technologie cible ... 116
5.2 ALIGNEMENT SEMANTIQUE ... 118
5.2.1 Distribution des espaces d'exécution ... 119

5.2.2 Alignement des méta-modèles de service ... 120
5.2.3 Alignement des opérations d'administration ... 124
5.2.4 Écart sémantique : Un modèle correct mais incomplet ... 126
5.3 APPROCHE « TOP-DOWN » ... 127
5.3.1 Service SAM ... 127
5.3.2 Service « legacy » enrichi ... 128
5.4 SYNTHESE ... 128

6. EXTENSIBILITE DE L'ENVIRONNEMENT D'EXECUTION ... 131

6.1 SEPARATION DES PREOCCUPATIONS DANS SAM-RT ... 131
6.2 GENERALISATION DE LA MECANIQUE DU *RUNTIME* ... 132
6.3 SYNTHESE ... 133

7. SYNTHESE : MODELE D'ETAT ... 135

CHAPITRE 4 - ÉVALUATION ET VALIDATION ... 139

1. PROJETS ... 139

1.1 SEMBYSEM ... 139
1.2 VALIDATEUR POUR FOCAS ... 142
1.3 DEMONSTRATEUR POUR LE PROJET SELECTA ... 143

2. ENVIRONNEMENT REPARTI : *ABSTRACT MACHINE* ... 145

2.1 DISTRIBUTION ... 145
2.1.1 Communication ... 145
2.1.2 Découverte ... 148
2.2 SPECIALISATION ... 149
2.3 NOTIFICATION ... 150
2.4 SYNTHESE ... 150

3. SERVICES DYNAMIQUES ET HETEROGENES ... 153

3.1 TESTS DE PERFORMANCES ... 154
3.1.1 Protocole et configuration des tests ... 154
3.1.2 Analyses des résultats ... 157
3.1.3 Comparaison avec d'autres technologies à service centralisées ... 158
3.2 SYNTHESE ... 160

4. ETENDU A D'AUTRES PREOCCUPATIONS : UNITE DE DEPLOIEMENT ET DEPOT ... 160

5. INFORMATIONS SUPPLEMENTAIRES ET SYNTHESE ... 167

CHAPITRE 5 - CONCLUSION ET PERSPECTIVES ... 169

1. CONTRIBUTIONS ... 170

1.1 ENVIRONNEMENT REPARTI ... 170
1.2 META-MODELE DESCRIPTIF ET PRESCRIPTIF D'ARCHITECTURES BASEES SUR DES SERVICES HETEROGENES ... 170
1.3 INTEGRATION ET INTEROPERABILITE DES SERVICES ... 171
1.4 EXTENSIBILITE DES PREOCCUPATIONS ... 171
1.5 UTILISATION DANS DES PROJETS ... 172
1.6 SYNTHESE ... 172

2. PERSPECTIVE DE RECHERCHE ... 174

2.1 L'ENVIRONNEMENT REPARTI ... 174
2.2 APPROCHE GLOBALE ... 174

BIBLIOGRAPHIE ... 177

ANNEXE A - ESPACE, ENVIRONNEMENT OU CONTEXTE 187
ANNEXE B - LEXIQUE ET DEFINITIONS 188
ANNEXE C - EXTENSIBILITE 189
ANNEXE D - INTERACTION *MANAGER/RUNTIME* 190
ANNEXE E - REGLE DE CRITERE QUALITE (SONAR) 191
ANNEXE F - EXEMPLES DE MESSAGE DU PROTOCOLE DE DECOUVERTE 194
ANNEXE G - LISTE DES PUBLICATIONS 195

Table des Illustrations

Figure 1 Contexte d'exécution des applications 15
Figure 2 Différentes technologies SOA pouvant être utilisées dans une même infrastructure..... 16
Figure 3 Administration d'une application hétérogène 17
Figure 4 Répartition d'une application..... 17
Figure 5 Types de modifications de l'architecture par les acteurs..... 18
Figure 6 Cycle de vie minimal du développement 22
Figure 7 Cycle de vie du déploiement défini dans [OMGa06]..... 23
Figure 8 Cycle de vie simplifié d'une exécution d'orchestration..... 24
Figure 9 Cycle de vie simplifié d'une application 25
Figure 10 Classification de l'adaptabilité des applications en fonction des phases de GL [KSKC04] 29
Figure 11 Exemple d'adaptation dynamique 30
Figure 12 Les 4 aspects de l'auto-administration (traduction de [Keph03])..... 33
Figure 13 Degré d'autonomie (à partir de [Gane03])..... 34
Figure 14 Les trois techniques de placement du code d'adaptation [Bou08]..... 35
Figure 15 Modèle et Représentation [Fav04] 39
Figure 16 Conformité 40
Figure 17 Relation entre un modèle et son système [FEB06] 40
Figure 18 Validité 42
Figure 19 Correction..... 42
Figure 20 Transformation de modèle 43
Figure 21 Schéma de l'approche proposée par [MBJFS09]..... 44
Figure 22 Classification des modèles à l'exécution [VSG10] 45
Figure 23 Méta-modèle Composant de l'OMG par [TIB05] (corrigé)..... 48
Figure 24 Méta-modèle simplifié de Fractal dans MIND (à partir de l'ADL) 49
Figure 25 Construction de l'assemblage au développement..... 50
Figure 26 Construction de l'assemblage à l'exécution..... 51
Figure 27 Conteneur et aspects non-fonctionnels..... 52
Figure 28 Patron d'interaction du SOC..... 57
Figure 29 Mécanismes d'un SOA [End04] 58
Figure 30 WS : technologies utilisées..... 61
Figure 31 Les trois couches de réutilisations [End04] 62
Figure 32 Orchestration 63
Figure 33 Composant Orienté Service défini par [CerT04] 66
Figure 34 Composant orienté service dans iPOJO 67

Figure 35 Composant orienté service dans SCA 69
Figure 36 Cycles de vie dans l'approche SAM 78
Figure 37 Architecture de l'environnement SAM 80
Figure 38 Groupe d'équivalence 83
Figure 39 Spécification SAM des services de l'exemple 86
Figure 40 Concept de spécification 86
Figure 41 Concept d'Implémentation 87
Figure 42 Exemple d'implémentations possibles pour la spécification "UI" 87
Figure 43 Concept d'Instance 88
Figure 44 Exemple de problème d'identification 89
Figure 45 Méta-modèle de SAM CORE 90
Figure 46 Cohérence dans l'environnement SAM 92
Figure 47 Dépendances dérivées 96
Figure 48 Méta-modèle SAM CORE à l'exécution 99
Figure 49 Exemple de sélection par navigation 100
Figure 50 Méta-modèle de *SAM-RT* 101
Figure 51 Espaces d'exécution : pont 103
Figure 52 Boucle de réification 104
Figure 53 Environnement d'exécution distribué 104
Figure 54 Exemple de répartition des technologies dans une infrastructure distribuée 105
Figure 55 Exemple de topologies 106
Figure 56 Recouvrement de protocoles de découverte 107
Figure 57 Exemple de deux *SAM-RT* interconnectées via leur AM 109
Figure 58 Identification d'un même service par plusieurs chemins 109
Figure 59 Distribution de services de SOA centralisés 110
Figure 60 Méta-modèle de l'*Abstract Machine* avec les trois registres de *SAM-RT* 111
Figure 61 Partie générique de SAM-CORE-RT 114
Figure 62 Validité : SCM 115
Figure 63 Correction : SCM 116
Figure 64 Architecture de SAM-CORE-RT 117
Figure 65 Principe d'interopérabilité de SAM-RT schématisé selon une topologie réseau 118
Figure 66 Exemple de SAM-RT* 119
Figure 67 org.osgi.service.ws.addressing.Endpoint défini et utilisé dans [BSSG08] 123
Figure 68 org.osgi.service.ws.addressing.MessageContent défini et utilisé dans [BSSG08] 123
Figure 69 Diagramme de transition d'état d'un bundle définit dans [OSGia] 124
Figure 70 Cycle de vie des Instances de composant (figure 57 de [EscT08]) 125

Figure 71 Cycles de vie d'un service SAM 126
Figure 72 Architecture d'un *Runtime* 132
Figure 73 Architectures d'un *Runtime* d'intégration 133
Figure 74 Exemple d'une SAM-RT étendue 134
Figure 75 Bloc de contrôle : manageur d'application 137
Figure 76 Architecture simplifiée du projet ITEA SEMbySEM 140
Figure 77 Schéma du démonstrateur pour SEMbySEM 141
Figure 78 Schéma logique du démonstrateur 142
Figure 79 Méta-Modèle du Composite SELECTA pour SAM-CORE 144
Figure 80 Diagramme de séquence d'un appel distant sur l'objet du service HelloWorld 147
Figure 81 Topologie de l'exemple de test 148
Figure 82 Exemples de proxies et de servants spécifiques 150
Figure 83 Consommation mémoire en fonction du nombre d'instances créées 157
Figure 84 Courbe de consommation mémoire pour 4000 instances de iPOJO (à gauche) et de SAM (à droite) 157
Figure 85 Temps moyen de 10 appels en fonction du nombre d'instances 157
Figure 86 Consommation mémoire en fonction du nombre d'instances créées 158
Figure 87 Temps moyen de 10 appels en fonction du nombre d'instances 159
Figure 88 Temps moyen de 10 appels en fonction du nombre d'instances 159
Figure 89 Un méta-modèle du concept d'unité de déploiement dans SAM 161
Figure 90 Exemple DeploymentUnit-RT 162
Figure 91 Un méta-modèle du concept de dépôt dans SAM 163
Figure 92 Exemple de modèles d'état 164
Figure 93 Quelques métriques sur l'ensemble du projet 167
Figure 94 Réalisation de la thèse 175
Figure 95 Architecture de l'environnement SAM 176
Figure 96 Espace, Environnement ou Contexte 187

Table des Tableaux :

Table 1 Comparatif de technologies à Composant [CerT04] 53
Table 2 Comparatif entre modèles à composant orienté service 69
Table 3 Caractéristiques de Rainbow 72
Table 4 Caractéristiques de JADE 72
Table 5 Caractéristiques de WComp 73
Table 6 Résumé des définitions de clés et valeurs des propriétés de groupe 84
Table 7 Comparaison partielle entre composant à service et composant orienté service 85
Table 8 Identificateur des concepts du méta-modèle de SAM CORE 89
Table 9 Alignement sémantique pour des services "legacy" 121
Table 10 Alignement sémantique pour des services importés 122
Table 11 Quelques métriques sur l'AM 151
Table 12 Quelques métriques sur SAM-CORE-RT (partie abstraite) 153
Table 13 Quelques métriques sur la SCM iPOJO 154
Table 14 Etats supportés par les concepts en fonction des extensions 164
Table 15 Caractéristiques de SAM-RT 173

Chapitre 1 - Introduction

L'apparition de l'approche orientée service (*Service-Oriented Computing* ou SOC) a changé le contexte d'exécution des applications du monde industriel. En effet, cette approche promeut l'abstraction des implémentations des fonctionnalités d'un logiciel sous la forme d'éléments logiciels appelés *services*. Ces *services* permettent d'abstraire d'un côté, la technologie utilisée pour les implémenter et de l'autre, ils permettent de s'abstraire de la distribution spatiale des *services*. En résumé, cette approche permet de composer des applications à partir d'éléments distribués et hétérogènes changeant ainsi profondément le contexte d'exécution des applications (cf. Figure 1).

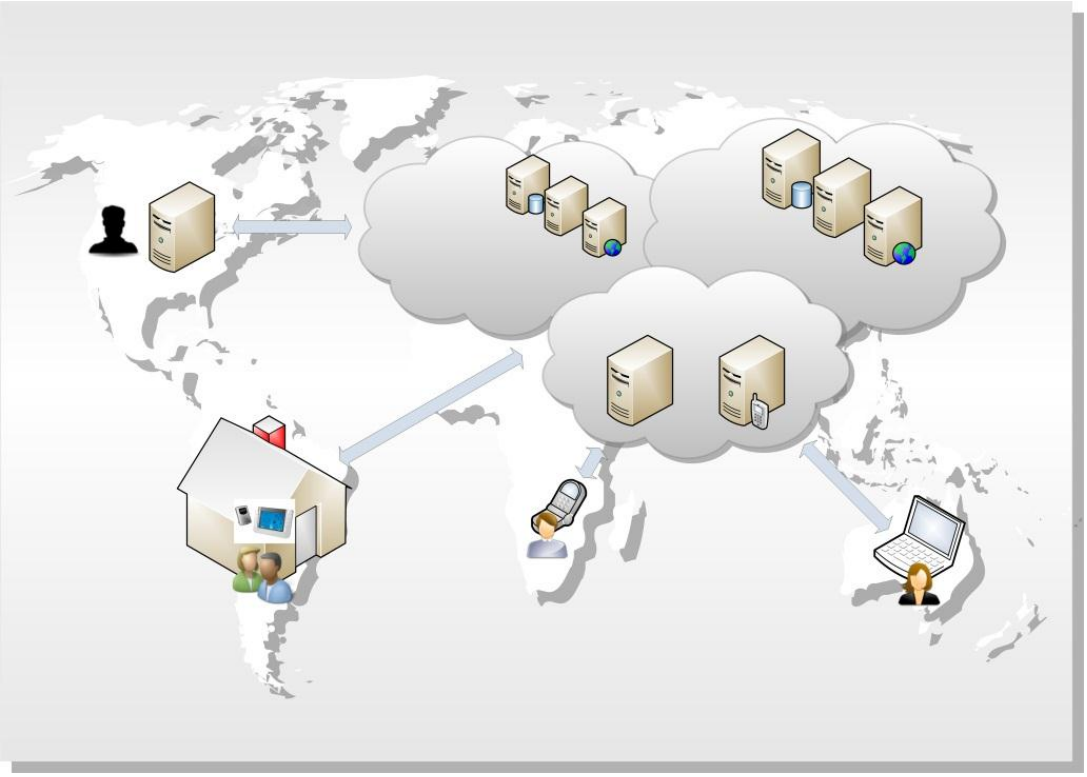


Figure 1 Contexte d'exécution des applications

Cette approche est maintenant utilisée par les industriels depuis au moins une dizaine d'année. Depuis peu, de nouveaux besoins émergent et guident le monde industriel vers le marché de l'informatique ubiquitaire (*ubiquitous computing*) qui consiste à intégrer dans une application des équipements informatiques (*device*) disséminés dans le monde réel. Nous commençons à voir, par exemple, des produits domotiques tels que des télévisions UPnP qui recherchent sur le réseau des serveurs de médias, ou des volets roulants DPWS pouvant être ouverts ou fermés via le Web. Ces équipements ont toutes les propriétés d'un service et sont traités comme tels dans les applications. Cependant ces nouveaux équipements imposent d'« ouvrir » les plates-formes au monde réel (*context aware*) et de gérer leurs propres protocoles. Or, jusqu'à présent les applications étaient généralement définies de manière statique, c'est-à-dire qu'à la phase de développement, tous les éléments de l'application sont identifiés et liés entre eux, aucune variabilité n'étant autorisée durant l'exécution.

L'informatique ubiquitaire pose au moins deux défis pour la définition et l'exécution d'application. Le premier défi est d'ordre technologique et le deuxième est d'ordre comportemental. Pour illustrer ces deux probèmes nous allons définir l'exemple suivant :

Une entreprise XYZ souhaite ouvrir un service de multimédia dans lequel un client peut à tout moment acheter un film, une musique ou autre. L'entreprise s'engage à fournir à la demande le document (film, musique …) sous le format le mieux adapté à la plate-forme cible. Par exemple, l'utilisateur désire regarder un film sur son *smartphone* – via un logiciel client dédié – l'entreprise encode le film pour obtenir la meilleur résolution possible selon le débit (*edge*, 3G, 3G+) et les ressources disponibles du *smartphone* et retransmet le flux vidéo. La résolution et le choix de l'encodeur ne seront pas nécessairement les mêmes si la plate-forme cible est un ordinateur portable qui dispose de plus de ressources et potentiellement un meilleur débit (*wifi,* RJ45…). Cette entreprise souhaite aussi louer une passerelle mulimédia pour la maison. le but de cette passerelle est de découvrir les équipements domotiques Hifi, pour lire les documents en exploitant les équipements présents. Par exemple, un client possède un téléviseur UPnP de résolution 1920*1080 pixel et un ampli DPWS en DTS 5.1. Ces équipement sont découverts par la passerelle multimédia de l'entreprise XYZ. Le client peut alors télécharger les documents sur sa passerelle multimédia dans son format optimal pour son équipement ; la passerelle se chargera de configurer la télévision et l'ampli.

Pour cela l'entreprise dispose de serveurs de stockage, de serveurs d'application, de serveurs Web (interface client)…

Le défi technologique vient du fait que de nos jours il existe un grand nombre de technologies SOA ayant des objectifs différents. Dans l'exemple précédent, les serveurs de l'entreprise peuvent être interconnectés / intégrés dans le système par des services web (*Web Services* , CORBA, Jini …) ; ces services pouvant être eux-même implémentés avec des technologies de composants à service (Spring DM, OSGI…) ou composants orienté service (iPOJO, Service Binder…). Finalement les équipements domotiques et ubiquitaires sont souvent basés sur des technologies SOA comme UPnP, IGRS, DPWS…

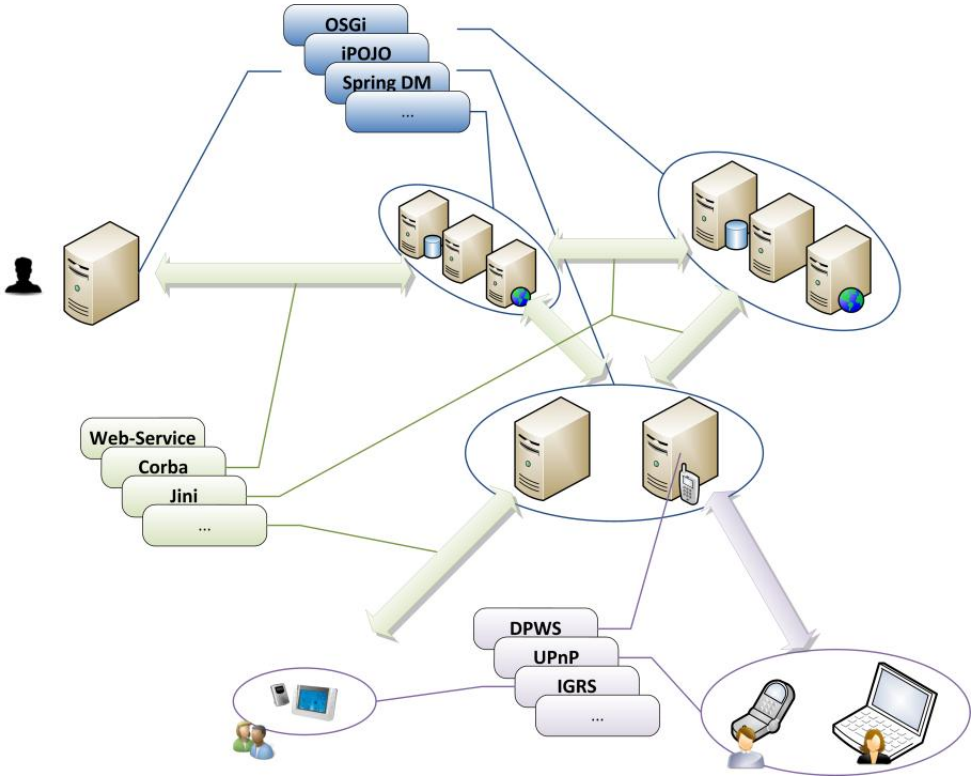


Figure 2 Différentes technologies SOA pouvant être utilisées dans une même infrastructure

Chaque SOA a des objectifs et des comportements propres. Certaines technologies servent à l'intégration et à la communication en abstrayant la technologie et la topologie ; d'autres servent à augmenter la réutilisabilité et la flexibilité en réduisant le coût ; et finalement certaines n'ont pour but que de pouvoir découvrir et utiliser des équipements ayant une disponibilité dynamique (cf. Figure 2).

Cette hétérogénéité est un des défis majeurs des applications ubiquitaires. Il existe un grand nombre de technologies SOA ayant chacune ses avantages et ses inconvénients. Les besoins actuels font qu'on les combine. Du fait de leur hétérogénéité, il est extrêmement difficile voire impossible d'administrer manuellement de telles applications.

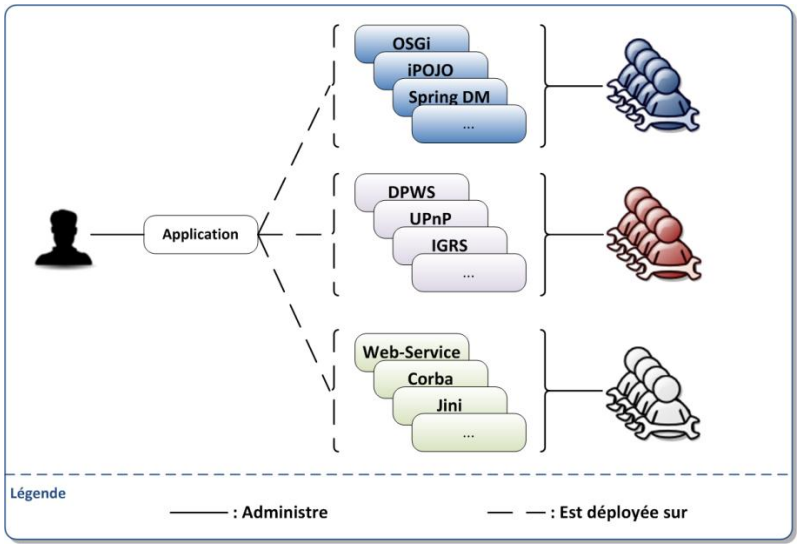


Figure 3 Administration d'une application hétérogène

Il est généralement nécessaire d'avoir plusieurs administrateurs s'occupant d'une ou plusieurs technologies ou d'un point de vue sur une ou plusieurs technologies (cf. Figure 3). De fait il faudrait avoir une équipe d'administrateurs uniquement pour administrer l'infrastructure indépendamment de l'application. De même la connaissance de l'application n'est pas uniquement éclatée entre les technologies mais aussi entre les machines (Figure 4).

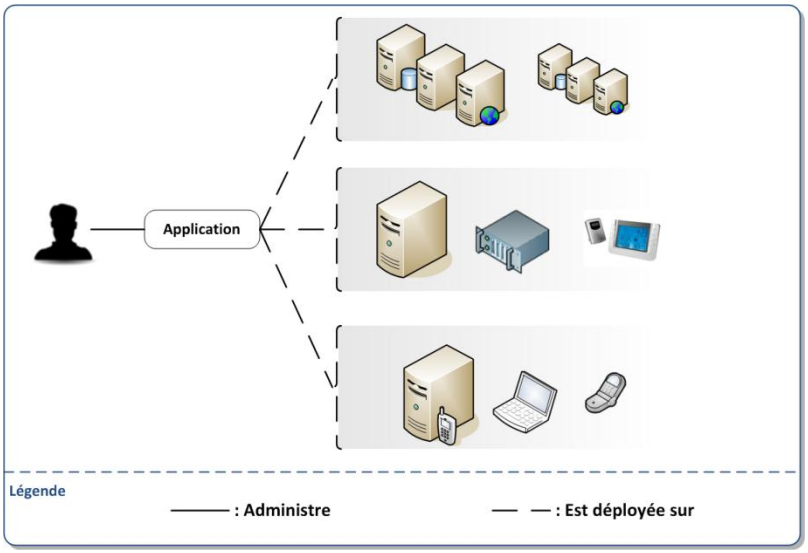


Figure 4 Répartition d'une application

Ces deux aspects (Figure 3 et Figure 4) rendent extrêmement difficile d'avoir une vision d'ensemble de l'application une fois déployée.

Le deuxième défi est lié aux comportements de ces applications à l'exécution. Jusqu'à présent, l'aspect dynamique des applications était réduit aux *liaisons retardées* et aux *chargements paresseux*. Cependant, l'informatique ubiquitaire se base sur des équipements ayant une disponibilité dynamique, c'est-à-dire que ces équipements peuvent apparaitre et/ou disparaître au cours de l'exécution de l'application. Par exemple un équipement peut être allumé puis éteint, cela peut aussi être un équipement mobile (ex. téléphone portable) qui rentre dans une zone « contrôlée » puis en sort. De fait comme le montre la Figure 5, dans ces applications, l'architecture de l'application peut être à la fois changée par l'administrateur en fonction des besoins de l'application ou du contexte d'exécution et à la fois par la disponibilité ou l'indisponibilité des équipements ubiquitaires.

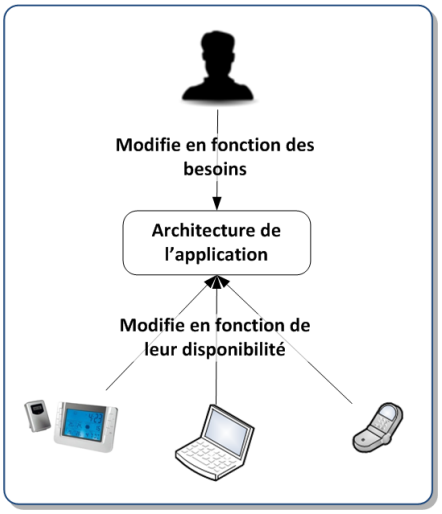


Figure 5 Types de modifications de l'architecture par les acteurs

Ces modifications de l'architecture peuvent être parfois conflictuelles. Il devient donc nécessaire d'une part de définir des applications qui définissent le comportement dynamique autorisé en incluant par exemple « des points de variabilité » dans l'architecture ; et d'autre part de définir des mécanismes d'adaptation qui modifient l'architecture en cours d'exécution conformément à la définition de l'application. Ces mécanismes nécessitent donc la vision globale de l'architecture de l'application. Or justement, nous venons de voir que les approches orientées service actuelles mènent à une fragmentation de la connaissance de l'architecture de l'application en cours d'exécution soit pour des raisons d'hétérogénéité technologique, soit de par la nature même de la technologie.

Le contexte illustré par la Figure 1 pose trois défis :

- Comment prendre en charge l'hétérogénéité technologique ;
- Comment prendre en charge la disponibilité dynamique des éléments de l'application à l'exécution ;
- Comment définir des mécanismes d'adaptabilité indépendamment du domaine métier et des technologies utilisées.

Il est intéressant de constater que ces défis sont traités par d'autres projets de recherche, par exemple : le projet JADE cherche à cacher l'hétérogénéité des serveurs pour pouvoir optimiser la disponibilité de ses applications sur des grappes indépendamment de la technologie. Le projet WComp adapte l'architecture en cours d'exécution en fonction de la disponibilité et de l'indisponibilité des équipements ubiquitaires qui la composent. Et finalement le projet Rainbow permet de mettre en place des mécanismes d'adaptabilité en basant le raisonnement sur une abstraction de l'architecture en cours

d'exécution, permettant de réutiliser ces mécanismes indépendamment du domaine et des technologies utilisées. Cependant, l'interconnexion des systèmes actuels avec des équipements ubiquitaires nécessite d'aborder ces trois défis simultanément.

En résumé, les applications actuelles doivent intégrer une multitude d'éléments hétérogènes dont l'architecture évolue dans le temps en fonction : des équipements présents, du contexte de l'utilisateur... Ces évolutions de l'application dépendent du contexte d'exécution. En d'autres termes l'architecture évolue dynamiquement avec ou non le consentement de l'administrateur. Il faut donc revoir la façon de concevoir et d'exécuter des applications pour que l'on puisse superviser l'évolution des applications et les adapter – si besoin est – pour que le but de l'application reste conforme à tous moments à nos attentes.

Cette problématique nécessite une vision globale couvrant les phases de conceptions, de développement, de déploiement et d'exécution des applications. Bien que nous exposons plus tard une proposition d'approche globale pour cette problématique, cette thèse propose un environnement d'exécution dirigé par les modèles pour la description et l'adaptation d'architecture à service hétérogène. C'est-à-dire, fournir une plate-forme qui :

- Cache l'hétérogénéité des technologies orientées service ;
- Réagisse à la disponibilité dynamique des éléments qui la composent ;
- Soit générique et spécialisable pour pouvoir définir des mécanismes d'adaptation en fonction d'un domaine métier.

Dans la suite de ce manuscrit, nous allons aborder l'état de l'art (Chapitre 2). Les quatre premières sections servent à définir le vocabulaire et les mécanismes qui seront abordés dans la thèse au travers de quatre aspects : le génie logiciel, le dynamisme et l'adaptabilité, l'utilisation des modèles et les paradigmes composant et service. La cinquième section de l'état de l'art décrit brièvement les projets Rainbow, JADE et WComp. Ces projets serviront à positionner nos travaux.

Le chapitre suivant aborde les contributions de cette thèse. (1) La première section de ce chapitre sert à positionner les travaux de cette thèse au sein de l'approche globale développée par l'équipe. Les sections suivantes exposent les différents éléments architecturaux pour la plate-forme : (2) le méta-modèle composant orienté service et (3) son utilisation à l'exécution, (4) la répartition de l'environnement, (5) l'intégration des technologies à service existantes et finalement (6) l'extensibilité du méta-modèle composant orienté service à d'autres préoccupations.

Le chapitre 4 valide et évalue l'environnement d'exécution réalisé aux travers de trois projets avant de détailler trois aspects que sont l'environnement distribué, l'évaluation en terme de mémoire et de performance en comparaison à d'autres environnements d'exécution composant orienté service que sont iPOJO, Tuscani et FraSCAti, et finalement valider l'extensibilité à l'aide de deux aspects liés au déploiement que sont les unités de déploiement et les dépôts d'unité de déploiement.

Chapitre 2 - État de l'art

La problématique soulevée par cette thèse couvre de nombreux domaines. Dans ce chapitre nous nous intéresserons à quatre domaines :

- Au génie logiciel : pour définir les différents aspects qui interviennent durant le cycle de vie d'une application de la conception à l'exécution.
- Au dynamisme et à l'adaptabilité : pour définir ce qu'est le dynamisme pour une application, quels sont les différents aspects de l'adaptation d'une application et quels sont les mécanismes nécessaires pour permettre l'adaptation des applications.
- A l'ingénierie dirigée par les modèles : pour décrire les mécanismes que nous utiliserons pour résoudre notre problématique.
- A deux paradigmes que sont les approches à composants et orientées service.

1. GENIE LOGICIEL

Le génie logiciel tire son origine d'une crise de l'informatique au milieu des années 60. En août 1967, le Comité des Sciences de l'OTAN, composé de scientifiques des divers états membres, créa un groupe d'étude dont la tâche était d'appréhender l'ensemble du domaine informatique. Ce groupe se concentra particulièrement sur le problème du logiciel. Fin 1967, les membres du groupe d'étude recommandèrent alors la tenue d'une conférence de travail sur le « Génie Logiciel » (*Software Engineering*) [Nato68]. Cette expression a volontairement été choisie car elle implique des fondements théoriques et des disciplines pratiques pour la conception de logiciel, aspects traditionnellement liés à l'ingénierie. En d'autres termes, la conception de logiciel doit être soumise à des règles approuvées et éprouvées dans le but d'améliorer sa qualité.

Selon l'arrêté ministériel du 30 décembre 1983, le « Génie Logiciel » désigne « l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi ».

Le domaine du génie logiciel est extrêmement vaste car il englobe tous les aspects de la production d'un logiciel, comme par exemple les processus métiers, la conception d'application, la vérification et la validation, jusqu'à la gestion de projet en terme de coût et de gestion d'équipe [FugA00]…

Dans ce chapitre nous nous concentrerons sur les activités du cycle de vie du logiciel fournissant des informations en vue de l'exécution. Dans un premier temps, nous aborderons les différentes phases nécessaires à l'exécution d'une application que sont : le développement, le déploiement et l'exécution. Ces différentes phases seront décrites succinctement et illustrées par un exemple concret. Nous conclurons ce chapitre en abordant les différentes « informations » qui sont utilisées dans ces phases / cycles de vie.

1.1 DEVELOPPEMENT

Durant le cycle de développement, l'application est définie, implémentée, testée… De nombreux cycles de vie de développement logiciel ont été proposés. Dans cette section, nous nous baserons sur l'approche SOMA (*Service-Oriented Modeling and Architecture*) [Ars08] [ZhZh09]. Cette approche cible les applications orientées services ; plus précisément elle est basée sur les services Web et par

conséquent la composition d'application se fait par orchestration (voir sections 4.2 et 4.3). L'approche SOMA découpe le cycle d'une application en 7 phases [Ars08]:

1. Modélisation du domaine métier (optionnel) ;
2. Gestion du projet de la solution : choix des méthodologies et de la gestion de projet ;
3. Identification : identification des services qui composent le flot de données ;
4. Spécification : définition des services qui composent l'application ;
5. Réalisation : cible les contraintes techniques (infrastructure) ;
6. Implémentation : développement et tests de l'application ;
7. Déploiement et supervision.

Dans ce chapitre, la phase 7 de SOMA ne fait pas partie du cycle de vie de développement d'une application, mais fait référence aux phases de déploiement et d'exécution. Ce cycle de vie peut être simplifié en quatre activités :

- La spécification de l'application ;
- La spécification des éléments de l'application en entité atomique ou composite ;
- Le développement des éléments atomiques ;
- La composition des éléments pour constituer l'application ou les composites.

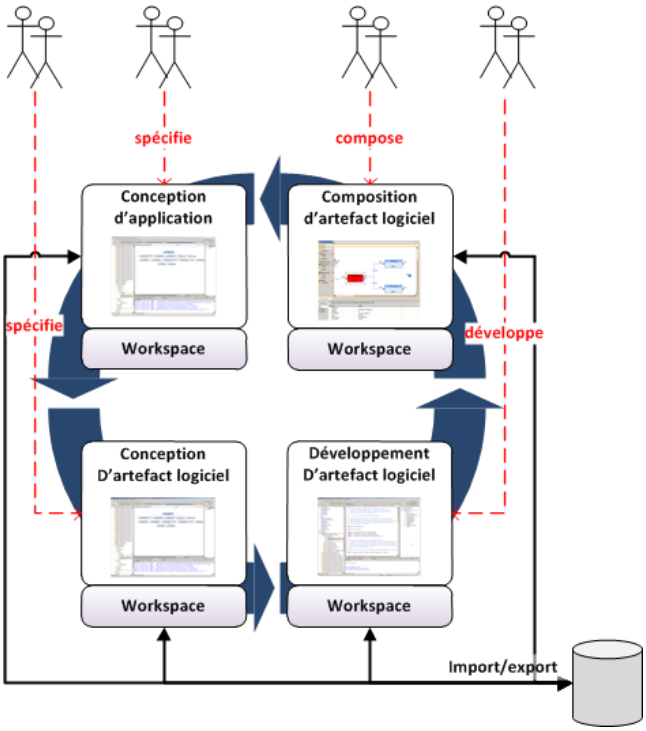


Figure 6 Cycle de vie minimal du développement

Notez que ces 4 activités font partie de l'ensemble minimal d'activité pour définir et développer une application. Cependant, il existe tout un ensemble d'activités ou de sous activités qui peut s'ajouter à ce cycle, comme la configuration de la sécurité ou des transactions…

Nous excluons volontairement la phase de déploiement qui, dans notre chapitre, possède son propre cycle.

1.2 DEPLOIEMENT

Il n'existe pas de consensus sur la définition de déploiement ni sur la portée de cette phase. Pour certains [Ars08], la phase de déploiement est une activité liée au développement, où celle-ci projette l'application sur un environnement physique comme étant la « finalité » du cycle de vie. Pour d'autres [CFAl98] [Hall99] [Dear07], les phases de déploiement et d'exécution sont une unique phase, car au final le déploiement – au sens : transfert, configuration, activation, mise à jour… – reflète le dynamisme à l'exécution de l'application. D'autres considèrent qu'il n'y a qu'un seul cycle couvrant les phases de développement, déploiement et exécution comme dans [PaHe06]. Et finalement certains, comme dans [OMGa06], considèrent que le déploiement est un cycle à part entière.

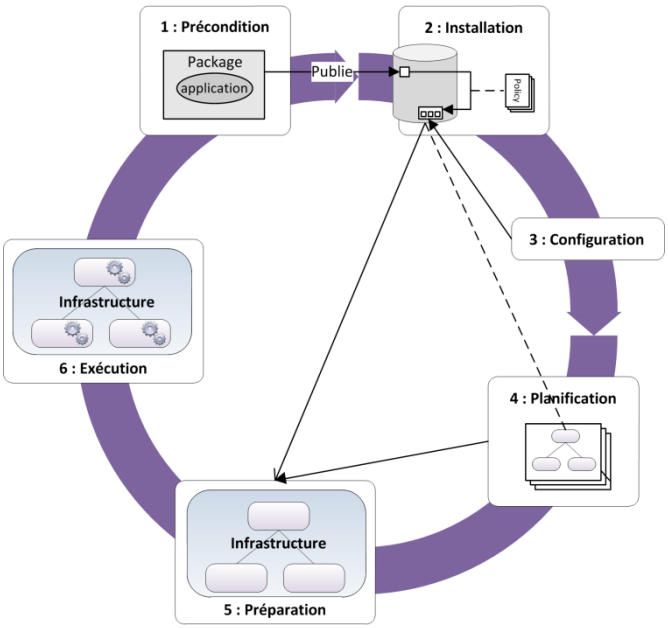


Figure 7 Cycle de vie du déploiement défini dans [OMGa06]

Il est intéressant de noter que le cycle défini dans la Figure 7 ne concerne que les applications distribuées basées sur des composants. Comme pour le cycle de développement, le cycle du déploiement dépend de la technologie, des paradigmes et de l'infrastructure. En d'autres termes il existe une multitude d'approches / de cycles possibles dépendants du but de l'application, des technologies, des approches utilisées et de l'infrastructure cible sur laquelle elle s'exécutera. Cette phase fait une « projection » de l'application sur une infrastructure en vue de son exécution.

Le déploiement est la phase de projection / synchronisation de la phase de conception à la phase d'exécution.

1.3 EXECUTION

La phase d'exécution est pour la plupart des approches la phase finale du cycle de vie de l'application juste avant la désinstallation. Prenons l'exemple d'une orchestration (cf. Figure 8) : l'application est définie dans la phase de spécification, les composants logiciels (services) sont alors développés et déployés sur différents environnements d'exécutions (contexte d'exécution des services). Le modèle de l'application est transféré à un orchestrateur / moteur d'orchestration (contexte d'exécution de l'application) qui l'interprète et l'initialise (création de la chaine d'activité). Lors de l'exécution de l'application, l'environnement d'exécution de l'application orchestre les activités. En pratique, bien qu'une orchestration autorise les propriétés de *liaison retardée* et de *chargement*

paresseux, l'exécution de l'application n'en reste pas moins statique. En effet, exceptée l'intervention d'un administrateur, l'architecture de l'application est immuablement identique de l'activation à la désactivation de l'application.

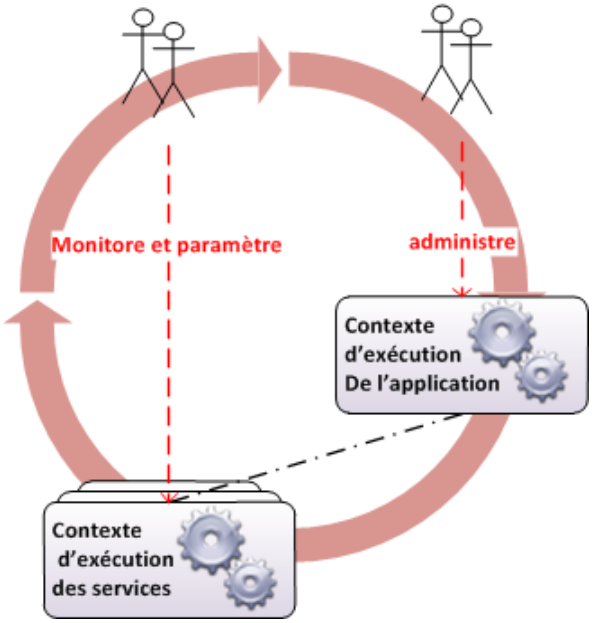


Figure 8 Cycle de vie simplifié d'une exécution d'orchestration

Il est intéressant cependant de noter que certains moteurs d'orchestration autorisent la substitution dynamique des services ; c'est-à-dire qu'ils autorisent la modification dynamique de la configuration de l'architecture grâce au couplage lâche de l'approche orientée service (cf. section 4.2 de ce chapitre).

Cependant, le monde industriel cherche, à juste titre, à fiabiliser et à sécuriser l'exécution des applications ; or le « dynamisme » tel que la substituabilité est une source d'erreur car il augmente le nombre de scénarii et d'architectures à tester est donc à priori moins fiable qu'une architecture dite statique. Par conséquent, dans la quasi-totalité des systèmes actuels, les activités du cycle de vie de l'exécution se résument à la supervision et aux modifications des paramètres fonctionnels (cf. Figure 8) ; les mises à jour et autres modifications faisant partie du cycle de vie du déploiement.

1.4 CREATIVITE, CONNAISSANCE ET CHOIX

Dans le cycle de vie d'une application, nous pouvons distinguer trois concepts : la C*réativité*, la *Connaissance* et les *Choix*.

La ***Créativité*** est la capacité de **créer** des **solutions** pour un problème donné. Cette capacité est le propre de l'homme. Par exemple développer un composant logiciel nécessite de la C*réativité* pour implanter les fonctionnalités.

La ***Connaissance*** est constituée d'informations implicites, explicites et contextuelles ainsi que « des liens de cause à effet ». Par exemple, il est nécessaire de connaître le ou les protocoles/conventions/spécifications pour implanter un composant dans une technologie donnée. Autre exemple, il peut être nécessaire de connaître la distribution de l'application afin de *packager* (empaqueter) ces éléments en vue du déploiement. La *Connaissance* est un concept extrêmement

vaste, nous nous limiterons donc par la suite à deux sous-types de *Connaissance* : la *Connaissance* de l'application (immuable et abstraite) et la *Connaissance* de l'exécution (dynamique et concrète).

La *Connaissance de l'application* correspond dans notre cas à la définition (modèle) de l'application en termes fonctionnels et non-fonctionnels. Cette *Connaissance* est établie lors du cycle de vie de développement.

La *Connaissance de l'exécution* correspond à la connaissance de l'état courant de l'environnement d'exécution.

Le ***Choix*** est la capacité de sélectionner, en fonction des *Connaissances,* une ou plusieurs **solutions** par rapport à un problème donné. Par exemple, la façon d'implémenter un composant logiciel dépend en partie de la technologie cible, mais aussi de choix personnels (liés à la Créativité) ; en effet, certaines solutions vont privilégier la vitesse d'exécution alors que d'autres vont privilégier l'empreinte mémoire. En l'absence de contrainte, le développeur fera un choix en fonction de sa *Créativité* et de sa *Connaissance*. De même si une plate-forme d'exécution doit sélectionner un service parmi plusieurs, en l'absence de connaissance et de critères, alors celui-ci fera un choix arbitraire : par exemple le premier trouvé. Cependant il pourrait « choisir » un service en fonction des *Connaissances* qu'il a du système en exécution, par exemple sélectionner un service qui n'est pas utilisé par d'autres clients.

1.5 SYNTHESE

Il n'y pas de consensus sur la définition des phases et cycles de vie d'une application. Cependant, nous trouvons trois phases – plus ou moins similaires – communes à tous les cycles de vie d'une application que sont le développement, le déploiement et l'exécution. Ces trois phases sont caractéristiques de la production d'une application.

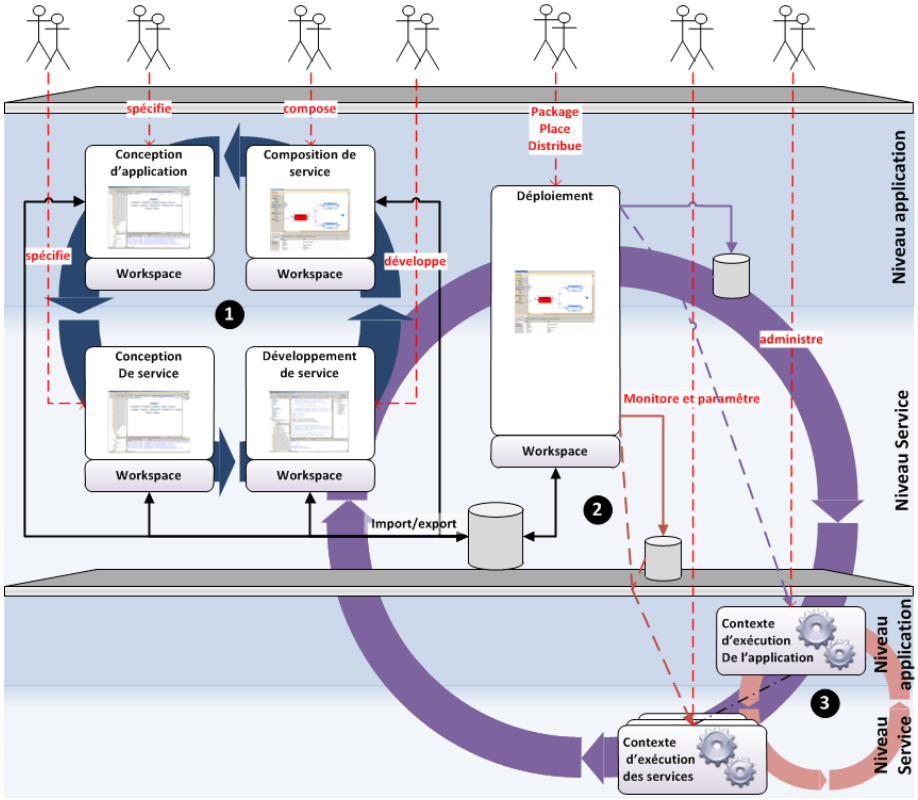


Figure 9 Cycle de vie simplifié d'une application

La **phase de développement** nécessite de la *créativité* pour résoudre un problème donné : l'application. Lors du développement, des *choix* (architectures, technologies…) sont effectués en fonction des besoins identifiés de l'application (*connaissance de l'application*).

La **phase de déploiement** fait la projection de l'application sur l'infrastructure en fonction de la *connaissance* de l'application et de l'infrastructure.

La **phase d'exécution** est très généralement une phase de supervision, les choix faits lors du déploiement ou au développement ne sont pas remis en cause, si ce n'est par un administrateur humain pour une mise à jour.

Nous voyons à travers les différents exemples que lors du cycle de vie de développement, les aspects de *Créativité*, de *Connaissance* et de *Choix* sont nécessaires. Dans l'approche orientée Service, les **spécifications** des services et des applications sont transmises au cycle de vie de l'exécution. Cependant dans les plates-formes à service actuelles, cette connaissance n'est très souvent exploitée que dans l'établissement des communications et les choix consistent à vérifier que les services sélectionnés sont conformes aux contraintes (SLA, propriétés non-fonctionnelles,…). Ces plates-formes gèrent peu (*liaison retardée* et/ou *chargement paresseux*) ou pas du tout l'aspect dynamique de l'exécution. Or les besoins actuels nécessitent de faire des choix en fonction des évènements qui surviennent à l'exécution, ce qui implique plus de *Connaissances* à l'exécution et la possibilité de modifier l'architecture courante.

2. DYNAMISME ET ADAPTABILITE

Internet a profondément bouleversé le monde informatique. En effet, ce réseau planétaire est devenu indispensable pour les entreprises car il permet à la fois d'interconnecter leurs infrastructures dispersées dans un monde physique, et d'offrir de nouveaux services. De ce point de vue, Internet a révolutionné les technologies de l'Information car il a permis de dissocier le monde logique du monde physique.

Parallèlement à cette dématérialisation, le monde industriel se tourne vers le marché de l'informatique ubiquitaire/ambiant (*ubiquitous computing* [Weis91]) qui consiste à intégrer dans une application des équipements informatiques disséminés dans le monde réel ; en d'autres termes : matérialiser un service logique par un équipement physique. L'émergence de ce marché est due au progrès des réseaux sans fil et de l'électronique qui ont permis d'embarquer de la logique dans des équipements courants. L'exemple le plus frappant est le téléphone portable qui, de nos jours, a de multiples fonctionnalités (appareil photo, baladeur, télévision,…). La liste de tels équipements est longue et ceux-ci peuvent être considérés comme des objets intelligents et communicants [Myer00].

Le rapprochement d'Internet et de l'informatique ubiquitaire ouvre la voie à de nouvelles applications très prometteuses. Cependant, ce rapprochement pose un certain nombre de défis. L'un des principaux d'entre eux est la capacité des plates-formes d'exécution à s'« ouvrir » au monde réel (*context aware* [ADBAl99]) et de gérer les protocoles de ces nouveaux équipements. En effet, ces nouvelles applications ne peuvent pas être dans un contexte fermé, elles doivent interagir « dynamiquement » avec les équipements disséminés dans le monde réel.

Dans ce chapitre nous allons définir ce que l'on entend par « dynamisme », « disponibilité » et « adaptabilité ». Ensuite, nous exhiberons une des approches permettant d'automatiser l'adaptation dite « dynamique » des applications : l'« informatique autonomique » (*Autonomic Computing*) [Horn01].

2.1 ADAPTABILITE ET DISPONIBILITE : DYNAMIQUE VS. STATIQUE

Les nouvelles applications issues du rapprochement d'Internet et de l'informatique ubiquitaire nécessitent l'ouverture du contexte d'exécution des plates-formes au monde réel. Or ce contexte d'exécution varie au cours du temps. Par exemple les équipements ubiquitaires sont, par nature, fortement dynamiques – c'est-à-dire que leur disponibilité varie au cours du temps – car ils peuvent apparaitre et/ou disparaître à tout instant : un téléphone portable peut être allumé puis éteint, il peut aussi rentrer dans une zone « contrôlée » puis en sortir. De fait, les applications doivent nécessairement gérer ce type de comportement.

En d'autres termes, dans l'informatique ubiquitaire, le dynamisme est caractérisé par la disponibilité des équipements. Par conséquent l'architecture des applications ubiquitaires varie en fonction de la disponibilité de ces équipements. En réponse à ce dynamisme dans l'architecture, il faut parfois adapter l'application pour garantir le but fonctionnel ou des buts non-fonctionnels.

Il faut donc bien distinguer **l'adaptabilité** d'une application qui est une réponse à la variation dans le temps de l'architecture dû à la **disponibilité dynamique** des éléments logiques qui la composent.

2.1.1 Disponibilité

Humberto Cervantes définit la disponibilité dynamique de la manière suivante :

> *« La disponibilité dynamique est définie comme le fait qu'une instance ou une classe de composant puisse devenir indisponible ou disponible à tout moment pendant qu'une application qui l'utilise ou qui pourrait l'utiliser est en train d'être exécutée. »*
>
> Extrait de [CerT04]

Nous considérons cependant que cette définition est partielle. En effet, ce qui est important c'est la notion de gestion de cycle de vie. Le cycle de vie de l'exécution d'un équipement ubiquitaire est inaccessible à l'administrateur de l'application. Reprenons l'exemple du téléphone portable : un opérateur téléphonique peut géo-localiser un client à partir de son téléphone si et seulement si le téléphone est allumé et se trouve dans une zone couverte par l'opérateur. Dans cet exemple ce n'est pas l'opérateur qui prend la décision d'éteindre ou de sortir l'utilisateur de la couverture du réseau. Par conséquent, l'opérateur considère les téléphones comme des éléments logiciels ayant une disponibilité dynamique, ceux-ci pouvant apparaître ou disparaître arbitrairement durant le cycle de vie de l'application de l'opérateur.

A contrario, la disponibilité dite statique s'applique aux éléments logiciels dont le cycle de vie est maitrisée par l'administrateur de l'application. Par conséquent, le type de la disponibilité dépend du point de vue que l'on prend.

Nous définissons les différents types de disponibilité de la manière suivante :

> *« La disponibilité dynamique est définie comme le fait qu'un élément logiciel puisse devenir **indisponible** ou **disponible** durant l'exécution d'une application, et ce **indépendamment** de la volonté de l'administrateur de l'application. »*

C'est le cas du téléphone portable défini dans le paragraphe précédent. Les plates-formes permettant à des applications de dépendre de ce type d'éléments doivent nécessairement définir un ou des mécanismes de notification de la disponibilité ou de l'indisponibilité des éléments (cf. : OSGi [OSGi-s]).

> *« La disponibilité semi-dynamique est définie comme le fait qu'un élément logiciel puisse devenir **indisponible** ou **disponible** durant l'exécution d'une application, partiellement sous le contrôle de l'administrateur de l'application. »*

Prenons le cas d'un environnement d'exécution réifiant via des proxies des équipements ubiquitaires, dans le but de les intégrer dans une application. Ces équipements ubiquitaires ont une disponibilité dynamique vis-à-vis de l'application. Cependant, l'administrateur de l'application peut générer des proxies et peut aussi les détruire. L'administrateur, bien que soumis à la disponibilité dynamique des équipements, peut contraindre partiellement cet équipement, en le masquant de l'application. Cet élément a donc une disponibilité semi-dynamique. Les plates-formes permettant une telle chose doivent d'un côté définir un ou des mécanismes de notification de la disponibilité ou de l'indisponibilité des éléments, et d'un autre permettre l'accès à la gestion du cycle de vie (création, destruction, configuration…) des proxies.

> *« La disponibilité semi-statique est définie comme le fait qu'un élément logiciel puisse devenir **indisponible** ou **disponible** durant l'exécution d'une application, **totalement** sous le contrôle de l'administrateur de l'application. »*

C'est, par exemple, le cas de l'utilisation basique de l'environnement d'exécution OSGi. Les cycles de vie de tous les éléments logiciels de l'environnement d'exécution, et donc de ou des applications, sont gérés par un administrateur. La plate-forme d'exécution OSGi définit les mécanismes de notification de la disponibilité ou de l'indisponibilité de ces éléments (*Service* et *Bundle*) et fournit des opérations primitives d'administration du cycle de vie des composants (bundles).

> *« La disponibilité statique est définie comme le fait qu'un élément logiciel est **invariablement disponible** durant l'exécution d'une application, selon la volonté de l'administrateur de l'application.»*

La disponibilité statique est le cas le plus courant. C'est le cas des applications dont la composition en termes de composant (classe, …) est définie avant le déploiement ; modifier un des éléments logiciels qui compose l'application nécessite l'arrêt de l'application.

Pour garantir la cohérence d'une application à l'exécution utilisant des équipements dits dynamiques, la plate-forme d'exécution doit, en fonction de la disponibilité dynamique de ces équipements, **adapter** l'application au changement de contexte d'exécution. **Caractériser le dynamisme** des éléments logiciels d'un environnement d'exécution permet d'anticiper les points de variation possibles d'une application. En effet, cette connaissance permet de fournir à la plate-forme d'exécution les points de variabilité de l'application et donc de garantir la cohérence des évènements durant l'exécution de l'application.

Nous utiliserons par la suite le terme d'*élément logiciel dynamique* pour tout élément logiciel ayant une disponibilité dynamique ou semi-dynamique, et *statique* pour tous ceux qui ont une disponibilité statique ou semi-statique.

2.1.2 Adaptabilité

Dans [Ore96], P. Oreizy affirme que l'architecture [AtWo92] de nombreux systèmes, en particulier ceux qui ont une longue durée d'exécution ou qui ont une mission critique, évolue durant l'exécution et donc que ces systèmes ne peuvent pas être modélisés et analysés en utilisant des architectures statiques ; il propose en conséquence le concept d'architecture dynamique (*Dynamic Architecture*).

Dans [KSKC04], P.K. McKinley définit une classification de l'adaptabilité des applications en fonction des phases du cycle de vie d'une application que sont le développement, la compilation, le chargement et l'exécution (cf. Figure 10).

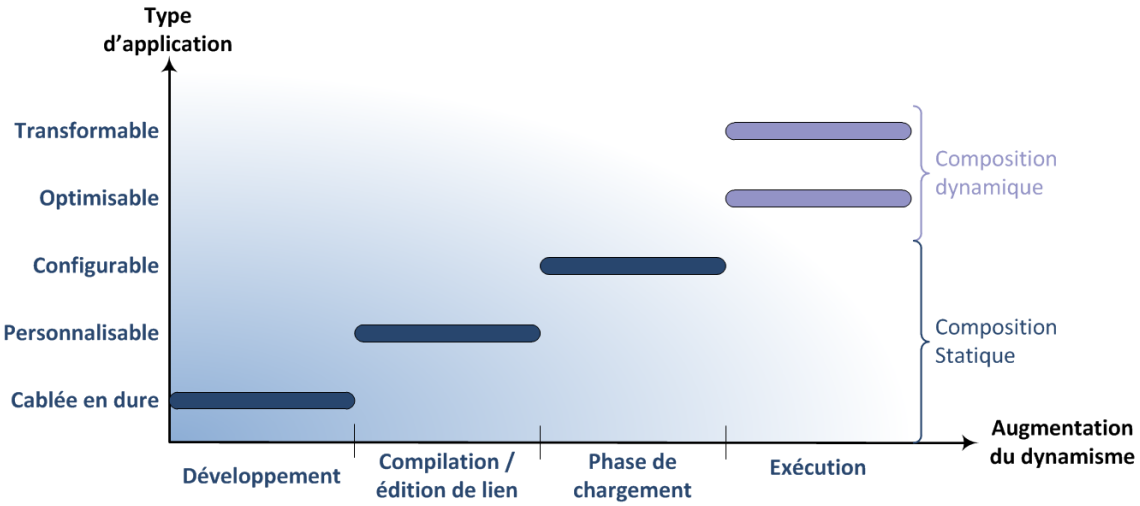


Figure 10 Classification de l'adaptabilité des applications en fonction des phases de GL [KSKC04]

Le niveau zéro de l'adaptabilité d'une application est l'application complètement définie (*Harwired*) lors de la phase de développement. A ce niveau zéro, s'adapter aux changements de l'environnement d'exécution impose de modifier le code de l'application. Le niveau 1 correspond aux applications dites « Personnalisables » (*Customizable*) qui sont adaptées à leur environnement d'exécution entre la compilation et le déploiement. Par exemple en utilisant des langages de programmation orientés aspect

comme AspectJ [KHHAl01]. Au niveau 2, les applications dites « Configurables » (*Configurable*) sont des applications dont l'adaptation au contexte de l'environnement d'exécution se fait juste avant le chargement des composants correspondants à l'exécution.

Ces trois niveaux sont des types d'adaptation statique de l'application au contexte de l'environnement d'exécution. En effet une fois l'adaptation effectuée, l'architecture des applications et leurs configurations ne sont plus modifiées à l'exécution modulo la maintenance d'un administrateur. L'adaptabilité de ces applications est dite « Statique ».

Les disponibilités dynamiques et semi-dynamiques des éléments logiciels ne sont pas autorisées dans ces trois niveaux ; car ceux-ci opèrent en amont de l'exécution et ne peuvent donc pas être réactifs aux changements du contexte d'exécution.

Le fait de modifier l'architecture d'une application à l'exécution n'implique pas nécessairement que cette modification soit une adaptation dynamique de l'application. P.K. McKinley définit qu'une application est adaptée dynamiquement si un « Compositeur » peut remplacer ou étendre les unités algorithmiques et structurelles en cours d'exécution sans avoir à arrêter et à redémarrer l'application. Il distingue deux types d'adaptation dynamique selon que le « Compositeur » peut modifier ou non la logique métier de l'application. Le premier type d'adaptation est l'optimisation (« Optimisable » / *Tunable*) ; cette adaptation interdit la modification du code métier et se concentre sur l'optimisation des paramètres d'exécution, fonctionnels ou non, en fonction de l'évolution du contexte d'exécution. Le second type d'adaptabilité dynamique est la transformation (« Transformable » / *Transformable*) ; cette adaptation autorise la modification de l'architecture ainsi que du code métier de l'application, comme par exemple substituer un composant de l'application par un autre durant l'exécution. L'intergiciel Open ORB [BCAl01] en est un exemple ; en effet il permet la substitution dans une application d'un composant par un autre sans avoir besoin de la redémarrer.

L'utilisation d'éléments logiciels dynamiques dans une application nécessite que l'application supporte l'adaptabilité de type transformable. L'exemple présenté dans la Figure 11 montre cette implication.

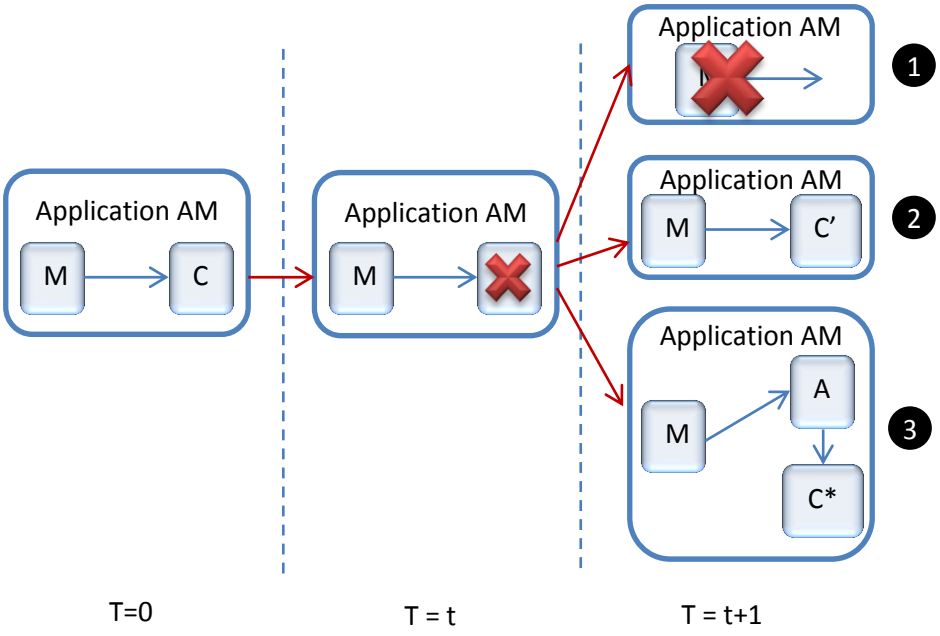


Figure 11 Exemple d'adaptation dynamique

Le but de l'application AM est d'afficher la météo, pour cela elle est constituée de deux éléments logiciels : M qui affiche la météo et C le capteur qui envoie les informations météo. A une date « t » du

temps d'exécution l'élément capteur C tombe en panne. Dans une application dite « statique », un tel scénario mène inexorablement à une défaillance de l'application AM (cf. : cas 1 de Figure 11).

Dans le cas d'application dite « dynamique », un tel scénario correspond au départ d'un élément logiciel ayant une disponibilité dynamique. Dans un tel cas de figure il y a plusieurs solutions possibles :

- 1) il n'y pas de capteur compatible disponible : l'application s'arrête ou se met en attente de l'arrivée d'un capteur compatible.
- 2) il y a un autre capteur compatible disponible (C'): ce capteur est utilisé dans l'application AM par l'élément M à la place de C.
- 3) il y a un autre capteur C* compatible au travers de l'élément d'adaptation A : l'application AM utilise le capteur C* par l'élément M via l'adaptateur A.

Il existe bon nombre de solutions pour un tel scénario ; cependant l'exemple ci-dessus suffit à mettre en évidence l'implication entre disponibilité dynamique d'un élément logiciel et l'adaptabilité des applications qui les utilisent.

Dans la suite de cette thèse, nous utiliserons la définition suivante :

> *« Une application dynamique est une application dont l'architecture est optimisable ou transformable durant son exécution ».*

2.1.3 Dynamisme : mécanisme non-transparent

La disponibilité et l'adaptation dynamique nécessitent des mécanismes (événements) notifiant les changements d'état des éléments logiciels. Plusieurs travaux ([KrMa07], [Ore99]) ont montré que la prise en charge de ces adaptations impacte directement l'architecture de l'application. En d'autres termes, afin de détecter les changements de contexte et d'adapter l'application en fonction de ces changements, l'application doit nécessairement être implémentée de manière à supporter l'adaptation de son architecture.

Dans le but d'adapter dynamiquement une application, l'environnement d'exécution doit avoir les mécanismes suivants :

- Introspection (Obligatoire) : l'environnement d'exécution doit fournir un ensemble d'informations concernant l'exécution, c'est-à-dire à la fois la Connaissance des applications exécutées, mais aussi la Connaissance du contexte d'exécution (CPU, mémoire utilisée, unités de déploiement installées…) ;
- Notification (recommandé) : l'environnement d'exécution doit notifier les changements qui ont lieu à l'exécution : arrivée et départ d'équipement dynamique, surcharge du CPU…
- Action (Obligatoire): l'environnement d'exécution doit fournir des opérations permettant d'adapter les applications et / ou l'environnement d'exécution. Dans [KrMa07], J. Kramer et J. Magee définissent un ensemble nécessaire d'opérations sur les éléments logiques d'une application. Ces opérations sont : la *création* et la *suppression* d'instance, la *configuration* des paramètres d'une instance et la *modification des liaisons* (dépendances) entre ces éléments.

Ces mécanismes sont nécessaires pour pouvoir adapter dynamiquement une application. Cependant ils ne sont pas suffisants. En effet, la notification permet d'informer qu'une modification a été effectuée ou est en cours ; l'introspection permet de connaître en détail l'état d'exécution fonctionnel et non-fonctionnel ; et finalement l'action permet d'agir sur l'environnement. Il manque toutefois l'analyse de l'état d'exécution et le choix des actions à effectuer.

En résumé, l'*Introspection*, la *Notification* et l'*Action* sont la mécanique qui permet l'adaptabilité des applications, mais la *Décision* / le *Choix* peut être implanté de différentes façons. Une de ces implantations est l'informatique autonomique.

2.2 INFORMATIQUE AUTONOMIQUE

Le terme informatique autonomique apparaît en 2001 dans un manifeste d'IBM [Horn01]. Le nom de ce paradigme tire son origine du système nerveux autonomique (*autonomic nervous system*). Le système nerveux autonomique gouverne le fonctionnement de notre corps, et ce, sans qu'on en soit conscient. Par exemple, Il régule notre pression sanguine, en déterminant la vitesse de battement de note cœur ; il surveille notre glycémie et notre niveau d'oxygène. En d'autres termes, il régule l'infrastructure de notre corps. Grâce à ce système, nous pouvons nous appliquer à d'autres tâches que celles de nous maintenir en vie.

2.2.1 Motivation

La motivation de l'informatique autonomique vient du fait qu'il y a, couramment, nécessité d'intégrer plusieurs environnements hétérogènes dans les systèmes informatiques des grandes entreprises, et de les étendre au-delà de leurs frontières. Cette nécessité introduit de nouveaux niveaux de complexité [Keph03] mettant à l'épreuve la capacité des meilleurs ingénieurs. Cette complexité grandissante repose sur plusieurs facteurs :

- les environnements d'exécution et les applications ;
- les enjeux économiques ;
- la popularisation des systèmes informatiques.

D'un côté, nous avons des ordinateurs de plus en plus puissants permettant de supporter des systèmes de plus en plus lourds ; nous voyons une multiplication des équipements mobiles tels que les smartphone, les GPS ou les netbooks qui se démocratisent ; l'explosion de l'internet qui interconnecte tous ces systèmes entre eux. D'un autre côté, les exigences des utilisateurs ne cessent de croître. Cet enchevêtrement de facteurs rend les nouveaux systèmes de plus en plus difficiles à intégrer, installer, configurer, régler et maintenir. Il existe de nombreux systèmes pour les serveurs d'applications, les serveurs Web ou les serveurs de base de données, qui nécessitent une réelle expertise pour être configurés et personnalisés. Leur degré de complexité est parfois tel que certaines personnes sont formées spécifiquement pour les gérer. Si cette complexité croit au rythme actuel, d'ici quelques années, même les plus qualifiés des ingénieurs se trouveront dans l'incapacité de gérer ces environnements [Keph05].

Ce constat a amené IBM [Horn01] à proposer la solution suivante : puisque l'homme n'est plus capable de s'adapter aux machines pour les utiliser, à celles-ci d'évoluer en fonction des enjeux.

Dans la suite de cette section, nous aborderons les enjeux de l'informatique autonomique ; la graduation entre ce qui est de l'administration manuelle de ce qui est autonomique ; puis finalement nous aborderons les différentes techniques d'implantation du comportement autonomique.

2.2.2 Enjeux de l'informatique autonomique

L'informatique autonomique présentée dans le manifeste d'IBM [Horn01], présente une vision en 8 éléments clés :

1. Pour être autonome, un système informatique a besoin de «se connaître».
2. Un système informatique autonomique doit pouvoir se configurer et se reconfigurer dans des conditions variables et imprévisibles.
3. Un système informatique autonomique ne se contente jamais du statu quo - il cherche toujours des façons d'optimiser son fonctionnement.
4. Un système informatique autonomique doit accomplir quelque chose qui ressemble à la guérison : il doit être en mesure de récupérer des événements ordinaires et extraordinaires qui pourraient causer un dysfonctionnement à certaines de ses pièces.
5. Un système informatique autonomique doit être un expert en auto-protection.

6. Un système informatique autonomique connaît son environnement et le contexte de son activité, et agit en conséquence.
7. Un système informatique autonomique n'a pas de raison d'exister dans un environnement hermétique (dans le sens d'une exécution dans un contexte « statique»).
8. Le cas ultime d'un système informatique autonomique est l'anticipation des besoins par une solution optimisée tout en conservant cachée la complexité.

Nous retrouvons dans ces éléments clés les requis pour qu'une application puisse être adaptée : pour qu'un système puisse être autonomique, celui-ci doit être introspectable (1) ainsi que son contexte d'exécution (2, 6 et 7) ; il est possible d'agir sur le système (2 et 6).

L'enjeu derrière le concept d' « informatique autonomique » est l'auto-administration connue sous le terme anglais : *self-management* ([Keph03], [PTDLK06], [PTDL07] ou [CGFP09]). L' « auto-administration » peut être décomposée selon 4 catégories [Keph03] ayant des enjeux différents : l'auto-configuration, l'auto-optimisation (3), l'auto-réparation (4) et l'auto-protection (5). Ces 4 catégories sont définies dans la Figure 12.

Concepts	Informatique actuelle	Informatique autonomique
Auto-Configuration	Les centres de données d'entreprises ont de multiples fournisseurs et de multiples plateformes. L'installation, la configuration et l'intégration de systèmes sont des tâches fastidieuses et sources d'erreurs.	La configuration automatique des composants et des systèmes suivent des politiques de haut niveau. Le reste du système ajuste la configuration automatiquement et en toute transparence.
Auto-Optimisation	Les systèmes ont des centaines de paramètres à définir manuellement suivant des réglages non-linéaires ; et leur nombre augmente à chaque nouvelle version.	Les composants et les systèmes ne cessent de rechercher des occasions d'améliorer leur performance et leur efficacité.
Auto-Réparation	La localisation des problèmes et leur cause dans les grands systèmes complexes peuvent prendre des semaines pour une équipe de programmeurs.	Le système détecte, diagnostique, répare automatiquement les logiciels et les problèmes matèriels.
Auto-Protection	La détection et la récupération des attaques et des défaillances en cascade sont manuels	Le système se défend automatiquement contre les attaques malveillantes ou des défaillances en cascade. Il utilise des mécanismes d'alerte rapide pour anticiper et éviter les pannes de l'ensemble du système.

Figure 12 Les 4 aspects de l'auto-administration (traduction de [Keph03])

Ces 4 catégories peuvent elles-mêmes être raffinées. Rendre un système autonomique n'est pas un but en soi ; il faut bien identifier les besoins et les requis. Prenons le cas d'une application dont la contrainte principale est la disponibilité ; dans ce cas l'aspect auto-optimisation n'est pas pertinente pour cette application, contrairement à l'auto-réparation qui permet d'améliorer / de garantir la disponibilité ou, potentiellement à l'auto-protection qui peut permettre d'éviter des indisponibilités dues à des actions malveillantes (ex. : déni de service).

2.2.3 Degré d'Autonomicité

Idéalement un système informatique autonomique [Horn01] devrait anticiper les besoins par une solution optimisée tout en conservant cachée la complexité du système (8). En pratique, mettre en place un tel système est extrêmement coûteux aussi bien financièrement qu'en performance. Dans le contexte actuel, mettre en place un tel système pour les applications est généralement déraisonnable, car disproportionné pour les besoins courants. Il faut bien distinguer les aspects et éléments du système qui doivent être supervisés et automatisés.

Automatiser un aspect d'administration ne signifie pas nécessairement d'exclure le facteur humain ; il existe plusieurs niveaux d'automatisation. Par exemple, on peut automatiser la détection des défaillances d'un système où l'administrateur diagnostique le problème et agit en conséquence ; on peut aussi automatiser le diagnostic ainsi que recommander une action.

	Niveau Basique	Niveau Géré	Niveau Prédictif	Niveau Adaptatif	Niveau Autonomique
Définition	• Multiples sources de données générées par le système	• Regroupement de données au travers d' outils de gestion	• Le système supervise, corrèle et recommande des actions	• Le système supervise, corrèle et agit	• Des composants gérés dynamiquement par des règles/politiques du métier
Besoins	• Besoin important de personnel hautement qualifié	• Le personnel analyse et agit	• Le personnel approuve et initie les actions	• Le personnel gère la performance en terme d'accord de contrat de niveau de service	• Le personnel se concentre sur les besoins du métier
Bénéfices		• Meilleure compréhension du système • Productivité accrue	• Réduit le besoin de personnel hautement qualifié • Les décisions sont plus précises et plus rapides	• Système agile et robuste avec une interaction humaine minimale	• Les règles métier régissent la gestion du système • Système agile et robuste
d'autonomie	Manuel				Autonomique

Figure 13 Degré d'autonomie (à partir de [Gane03])

La Figure 13 [Gane03] définit différents degrés d'autonomicité en fonction des besoins et des bénéfices apportés par cette automatisation / autonomisation.

2.3 ADAPTABILITE : TECHNIQUES D'IMPLANTATION

Il existe trois « types » de techniques d'implantation de l'adaptabilité (autonomique ou non).

La première technique consiste à **mélanger la logique d'adaptation et la logique de l'application** (cf. : Figure 14 schéma 1). Cette technique est la plus simple car l'adaptation est définie et implémentée en même temps que le code fonctionnel, en contradiction avec le principe de séparation des préoccupations. Conceptuellement, cette manière de faire signifie que l'adaptabilité est un besoin fonctionnel de l'application. Toutefois cette technique a pour avantage que le code d'adaptabilité est totalement spécifique et intégré à l'application et ne nécessite pas d'infrastructure spécifique. Mais le pendant est que la modification du comportement de l'adaptabilité nécessite la modification de ou des éléments d'une application complexifiant la maintenance de l'application. En résumé, cette technique est à employer dans le cas d'applications dont les besoins en performance sont importants et dont **le comportement dynamique** est **limité** et **standardisé**.

La deuxième technique consiste à **définir la logique d'adaptation dans le conteneur de l'application** (cf. : schéma 2 Figure 14). Cette technique est utilisée dans les plates-formes à composants (cf. : section 4.1). Cette technique respecte le principe de séparation des préoccupations, où le code d'adaptabilité est séparé du code fonctionnel. Par conséquent, la maintenabilité du code fonctionnel

n'est pas impactée par les modifications du code de l'adaptabilité. Cependant la réciproque est fausse : en effet le code d'adaptation placé dans le conteneur est spécifique à l'implémentation du composant car il supervise et agit sur la mécanique interne du composant. Cette technique permet d'avoir **une meilleure maintenabilité** que la première technique tout en **conservant une bonne performance** ; cependant cette technique requiert une **infrastructure spécifique** et le **code d'adaptabilité reste spécifique** aux éléments de l'application.

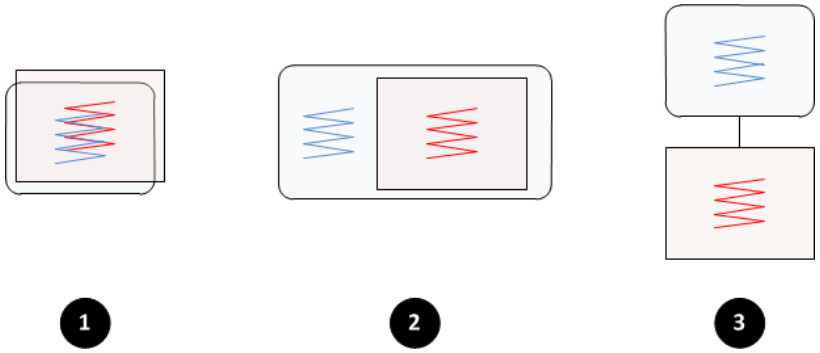


Figure 14 Les trois techniques de placement du code d'adaptation [Bou08]

La troisième et dernière technique consiste à **définir la logique d'adaptation séparément de l'application** (cf. : schéma 3 Figure 14). Comme dans la technique précédente, celle-ci suit le principe de séparation de préoccupation ; ce qui la démarque le plus est qu'elle découple totalement la logique d'adaptation de l'application. En effet, cette technique permet de s'abstraire de l'implémentation des composants de l'application ; car cette technique se base sur les interfaces d'administration des composants de l'application préalablement définies. L'avantage de cette technique est que le code d'adaptabilité de l'application est libéré de toutes « contraintes » liées à l'application (technologie, topologie, architecture…) ; cependant cette technique nécessite que les interfaces d'administrations nécessaires à l'adaptation soient définies au préalable.

2.4 SYNTHESE

Les besoins actuels en informatique évoluent vers des plates-formes s'adaptant aux capacités et aux contextes de l'environnement. Or jusqu'à présent, les applications étaient généralement définies de manière statique, c'est-à-dire que lors du développement, tous les éléments de l'application étaient identifiés et liés entre eux. L'aspect dynamique des applications était réduit aux *liaisons retardées* et aux *chargements paresseux*. Cependant les besoins actuels font appel à des équipements qui sont, par nature, fortement dynamiques. Par exemple un appareil peut être allumé puis éteint, cela peut aussi être un équipement mobile (ex. téléphone portable) qui rentre dans une zone « contrôlée » puis en sort. Il devient impératif d'assurer la conformité de l'évolution des applications dans un environnement d'exécutions dynamiques et ouvert.

En réalité le problème est plus profond : pour satisfaire les besoins actuels il faudrait avoir des environnements d'exécution qui s'adaptent aux changements de contexte et aux besoins non-fonctionnels. Or, le contexte d'exécution est un concept diffus, il faut le limiter aux aspects pertinents, ceux-ci pouvant varier au cours du temps. Définir une application statiquement en prenant en compte l'ensemble « pertinent » des contextes pour son exécution est très difficile car l'ensemble des contextes peut être infini et dépendre de l'installation courante. De plus l'ajout d'un contexte nécessite un nouvel incrément au cycle de développement empêchant ainsi l'adaptabilité dynamique.

Dans le cadre de cette thèse, les artefacts logiciels ont une disponibilité allant du statique au dynamique. Le but étant de fournir une représentation dynamique d'un système en exécution qui

permet de mettre en place des manageurs d'applications, et où ces manageurs assurent l'évolution des applications en adaptant l'architecture aux besoins et aux contraintes de l'environnement d'exécution.

3. MODELES DANS LE CYCLE DE VIE D'UNE APPLICATION

En novembre 2000, l'*Object Management Group* (OMG) propose une approche nommée MDA[1][MDA-O],[KWB03] qui a pour but de décrire (modéliser) les systèmes indépendamment des plates-formes (PIM[2]) pour ensuite les raffiner vers une ou plusieurs plates-formes spécifiques (technologie cible)(PSM[3]). Le MDA a une vision très réductrice du concept de modèle, qui n'est au final qu'une abstraction de l'architecture d'une application. De nos jours, l'utilisation des modèles va au-delà de la simple définition d'une architecture, et se retrouve dans de nombreux travaux [BJV04] [FR07] [KWB03]. L'ensemble des travaux basés sur la manipulation de modèles peut être regroupé sous le terme d'Ingénierie Dirigée par les Modèles (IDM) [FEB06]. Le but fondamental de l'Ingénierie Dirigée par les Modèles est de faire face aux problèmes (récurrents) de la complexité, de la croissance et de l'évolution rapide des systèmes logiciels.

Ce chapitre présente succinctement les concepts de base de l'IDM, la notion d'abstraction, les formalismes nécessaires à l'exploitation informatique des modèles et finalement le principe des modèles à l'exécution : *model@runtime*.

3.1 CONCEPT DE BASE

Le concept de *modèle* n'est pas propre à l'informatique et est bien antérieur à l'approche MDA [War79]. Même en ne se limitant qu'au domaine informatique, il n'existe pas de définition universelle et précise. Cependant, la définition de M. H. Lee (cf. la définition ci-dessous) exhibe la caractéristique commune à toutes les définitions du concept de modèle qui est la notion de représentation.

> *"A model implies a representation; i.e., a model imitates, resembles or stands for something else. This representational aspect of models is their most significant characteristic."*
>
> Extrait de [Lee00]

En d'autres termes, un modèle est la représentation / imitation de quelque chose. Dans [Lud03], J. Ludewig propose trois critères pour caractériser un modèle :

- Critère de représentation (*Mapping criterion*) : il existe un objet ou un phénomène « original » qui est représenté par le modèle.
- Critère de réduction (*Reduction criterion*) : toutes les propriétés de « l'original » ne sont pas représentées dans le modèle, cependant le modèle reflète quelques propriétés de l'original.
- Critère de pragmatisme (*Pragmatic criterion*) : le modèle peut remplacer « l'original » pour un usage donné.

Ces critères se retrouvent dans les définitions suivantes du concept de modèle :

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

---

[1] *Model Driven Architecture* : architecture dirigée par les modèles
[2] *Platform Independant Models* : modèles indépendants de la plate-forme
[3] *Platform Specific Models* : modèles spécifiques à une plate-forme

Selon cette définition, un modèle est une représentation simplifiée et réduite à un point de vue (aspect/préoccupation) d'un système courant. Ce modèle devrait pouvoir se substituer au système courant pour répondre aux questions sur ce point de vue. Cette définition est la combinaison des trois critères de [Lud03]: le modèle représente un système existant (Critère de représentation) ; c'est une simplification du système existant (Critère de réduction) et devrait pouvoir remplacer le système courant pour un but donné (Critère de pragmatisme).

> *"The system described by a model may or may not exist at the time the model is created. Models are created to serve particular purposes, for example, to present a human understandable description of some aspect of a system or to present information in a form that can be mechanically analyzed."*
>
> Extrait de [FR07]

Dans la définition ci-dessus, un modèle n'est pas nécessairement une description d'un système déjà existant ; il peut aussi être la spécification d'un système à développer. Ce modèle peut avoir pour but d'être une description compréhensible pour un humain sur un aspect précis du système, ou fournir des informations formalisées pour être analysé logiquement. Selon cette définition un modèle peut être utilisé dans un **but contemplatif ou productif**.

Nous voyons au travers de ces définitions qu'il n'existe pas qu'un seul « type » de modèle ; en effet un modèle a une nature [Fav04], un but [FR07]… Cependant, nous distinguons trois concepts inhérents que sont : le modèle, le système (décrit ou spécifié) et le formalisme.

Dans la suite de ce chapitre, nous allons, dans un premier temps, discuter de deux aspects des modèles dérivés des critères de réduction et de pragmatisme que sont l'abstraction et la séparation des préoccupations ; puis aborder le formalisme et finalement exhiber quelques exemples d'utilisation de modèle dans l'IDM.

3.2 ABSTRACTION ET PREOCCUPATIONS

Le principal but de l'Ingénierie Dirigée par les Modèles est de réduire la complexité des systèmes informatiques. Ce but est principalement atteint grâce à deux mécanismes : l'Abstraction - suivant le critère de réduction – et la séparation des préoccupations – suivant le critère de pragmatisme. Nous aborderons dans la suite de cette section ces deux mécanismes.

3.2.1 Abstraction : une notion de Granularité

Un modèle est par définition une représentation / une imitation. Cependant, selon le critère de pragmatisme, le modèle peut se substituer au système pour un point de vue précis qui peut être lui-même modélisé. Par exemple, une spécification de logiciel est un modèle du code à écrire et le code écrit est le modèle des actions à l'exécution de ce logiciel. Inversement suivant le critère de réduction, la logique d'un système est modélisée par son code, ce code peut être réduit (modélisé) selon un aspect comme par exemple la notion d'héritage dans ce système. Les notions de modèle et de système original sont donc relatives. Ces notions sont synthétisées par le diagramme de classe UML[4] de la Figure 15 extrait de [Fav04].

[4] *Unified Modeling Language* : c'est un langage de modélisation graphique définit par l'OMG. http://www.uml.org/

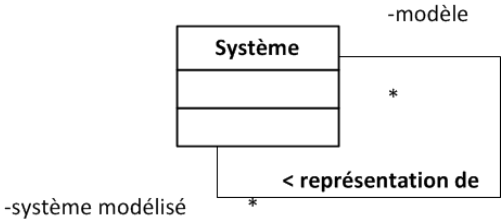


Figure 15 Modèle et Représentation [Fav04]

L'approche MDA proposée par l'OMG préconise la spécification d'un système par un modèle indépendant de la plate-forme cible (PIM) pour ensuite le raffiner par itérations successives vers un modèle spécifique à une plate-forme (PSM). Cet exemple exhibe une notion importante de la modélisation qu'est l'abstraction. Plus un modèle est raffiné, plus sa précision augmente et plus sa granularité devient fine. Inversement, plus un système est modélisé suivant le critère de réduction, plus sa granularité grossit, moins forte est sa précision. Dans le cas du MDA, le mécanisme d'abstraction permet de séparer la complexité liée à la problématique du logiciel de la complexité liée à la technologie utilisée pour l'implanter.

3.2.2 Séparation des préoccupations

Le deuxième mécanisme / approche permettant d'attaquer la complexité des systèmes logiciels est la séparation des préoccupations (*Separation of Concerns / SoC*) [Par72] [HL95] [TOHS99]. Le principe de cette approche peut se résumer selon la définition suivante :

> *« La "séparation des préoccupations" est une approche qui vise à regrouper le code correspondant à un même aspect et à rendre les aspects indépendants. »*
>
> Extrait de [DES02]

L'abstraction du PIM dans l'appriche MDA (cf. 3.2.1 de ce chapitre) permet de séparer la préoccupation du métier de la préoccupation technologique. Dans l'IDM, cette séparation peut s'appliquer à tous les points de vue ou aspects particuliers d'une application qui sont décrits par un modèle particulier. La notion même d'abstraction d'un système est nécessairement relative à un point de vue. Par conséquent, la séparation des préoccupations est une caractéristique intrinsèque de l'IDM correspondant aux critères de réduction et de pragmatisme.

Un système logiciel peut donc être décrit par un ensemble de modèles correspondant chacun à une facette de ce système. La combinaison des différentes facettes est toujours envisageable [Bez05]. Dans le cas inverse, où le modèle est prescriptif, la combinaison des différents modèles (préoccupations) se fait par la composition et le tissage de tous ses aspects. Dans ce dernier cas, il faut assurer la cohérence entre les différents modèles de la spécification, alors même que la composition et le tissage de modèle dans un but productif (génération de code) restent un défi courant.

En résumé, la séparation des préoccupations est une caractéristique intrinsèque à la modélisation. Un système peut avoir autant de modèles que de points de vue, et où l'union de tous les modèles pragmatiques d'un système est équivalente à ce dernier.

3.3 FORMALISME

Dans les sections précédentes nous nous sommes intéressés à l'aspect représentation et non à la nature même de la relation entre un système et son modèle. Un modèle peut être de nature mentale, physique ou logique [Fav04]. Cet aspect a son importance par le fait que dans le cadre de l'IDM, les modèles sont numériques, informatiques, pouvant être traités et exploités automatiquement. Comme toute donnée informatique, les modèles doivent donc être formalisés suivant un langage bien défini

dans le but d'être exploités. En d'autres termes pour qu'un modèle puisse être traité automatiquement, il faut qu'il soit défini dans un langage formel de modélisation.

Pour définir un langage de modélisation, il est nécessaire de spécifier tous les aspects du langage et par conséquent il devient lui-même modélisable. Comme tout système, un langage peut être représenté par plusieurs modèles (descriptifs ou prescriptifs). Ce modèle qui spécifie le langage de modélisation est appelé **méta-modèle** [Fav04]. La relation entre un modèle et son méta-modèle est équivalente à la relation entre un programme et son langage de programmation [Bez05].

Un langage se décompose en deux notions : la forme (syntaxe) et le sens (sémantique).

> *"A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer."*
>
> Extrait de [KWB03]

La relation entre le modèle et son méta-modèle s'appelle la **conformité**. Un modèle est dit conforme à son méta-modèle si les concepts des éléments du modèle et les relations qui les relient sont définis dans le méta-modèle et si les contraintes du méta-modèle sont respectées par le modèle.

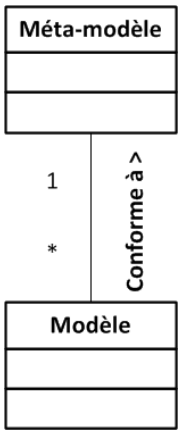


Figure 16 Conformité

Comme pour tout langage, un modèle peut être syntaxiquement correct (conforme) mais n'a pas nécessairement de sens, ou dans le cas d'un modèle descriptif peut être faux.

Faisons la synthèse entre la Figure 14 et la Figure 15, un système peut être représenté (modélisé) par un autre système suivant un formalisme défini par un méta-modèle. Quels critères permettent de dire que cette représentation est exacte ?

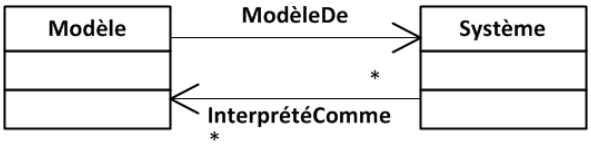


Figure 17 Relation entre un modèle et son système [FEB06]

Causalité : validité et correction

Dans [Jack02], M. Jackson expose le problème de la relation entre un modèle descriptif et la réalité ; la teneur de cet article est résumée dans [MFBC10] comme une relation de causalité (*causality*).

> *"Causality addresses the synchronization concern raised by Michael Jackson "*[Jack02]*"; it expresses both* when *the µ-relation is established, and* how *(if ever) it is maintained over time. Causality is either continuous (the relation is always enforced) or discrete (the relation is enforced at some given points in time)".*
>
> Extrait de [MFBC10]

De fait la relation ModèleDe dépend de la nature du modèle. Dans le cas d'un modèle prescriptif, celui-ci est transformé suivant une logique de raffinement jusqu'à un niveau « d'instanciation ».

> *"[…] we can use a model as a specification for an SUS" (*system under study*)" or a class of SUS. In this case, we consider a specific SUS valid relative to this specification if no statement in the model is false for the SUS."*
>
> Extrait de [Seid03]

Selon la définition précédente, dans le cas d'un modèle prescriptif, le système étudié est *valide* si aucune assertion dans le modèle (sa spécification) n'est fausse pour le système.

Inversement dans le cas d'un modèle descriptif, les aspects identifiés du système étudié sont représentés dans le modèle.

> *"We can use a model to describe an SUS" (*system under study*)". In this case, we consider the model correct if all its statements are true for the SUS."*
>
> Extrait de [Seid03]

Selon la définition précédente, un modèle descriptif est *correct* si toutes les assertions sont vraies pour le système étudié.

Relations

Les notions de *validité* et de *correction* ne permettent pas de relier les éléments du modèle aux éléments du système.

En effet, la validité ne définit pas comment un élément du modèle prescriptif est matérialisé dans le système étudié. Cette matérialisation peut se faire : manuellement par un développeur, automatiquement par un générateur ou par les deux. Les éléments d'un modèle prescriptif – formalisé selon son méta-modèle (et donc conforme à son méta-modèle) – sont matérialisés dans le système étudié. Notons que tous les systèmes ont un méta-modèle explicite ou implicite. La matérialisation dans le système étudié se fait selon les éléments du système ; ces éléments sont nécessairement être définis dans le méta-modèle – explicite ou implicite – du système étudié.

> La relation *« imageDe »* relie des éléments d'un modèle prescriptif à des éléments du système étudié.

Cette matérialisation se fait selon des *règles de matérialisation*. Dans le cas d'une génération ces règles sont formalisées. Dans le cas manuel, cette formalisation est beaucoup plus compliquée.

> Les règles de *matérialisation*, matérialisent des éléments du modèle prescriptif dans le système étudié conformément au méta-modèle du système. Par conséquent, les éléments du modèle prescriptif sont - par matérialisation - *« imageDe »* l'élément du système.

De même, la *correction* ne définit pas comment les éléments d'un système étudié sont identifiés et comment ils sont modélisés.

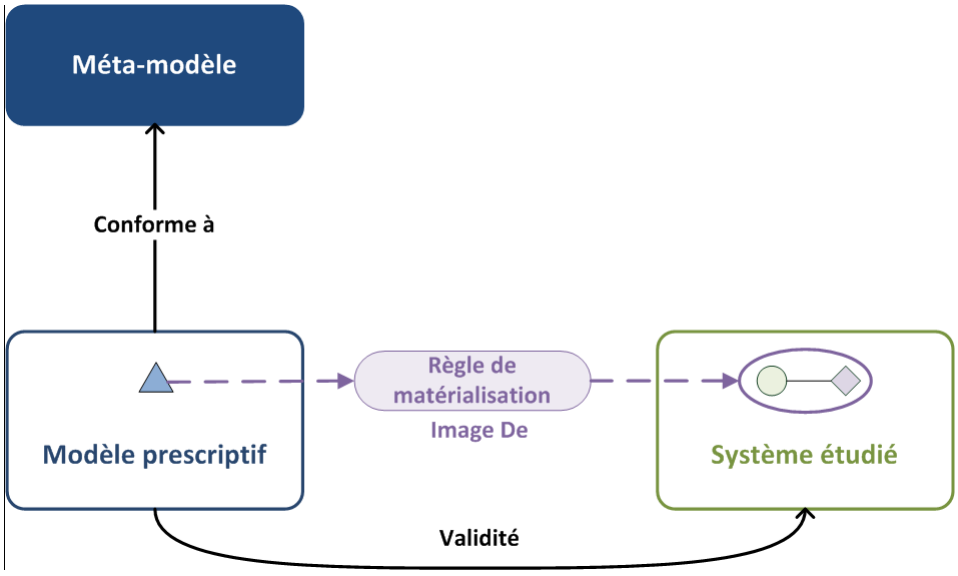


Figure 18 Validité

L'identification des éléments du système étudié se fait selon un point de vue. En effet il est inutile d'identifier des éléments du système si on ne cherche pas à les modéliser / représenter. Par conséquent les règles définissant l'identification sont liées aux règles de représentation du modèle descriptif.

Ces règles de *représentation* modélisent les éléments identifiés dans le système étudié conformément au méta-modèle du point de vue cible. Plus simplement, un élément identifié d'un système étudié sera représenté dans le modèle descriptif selon sa règle de *représentation* conformément au méta-modèle du point de vue X.

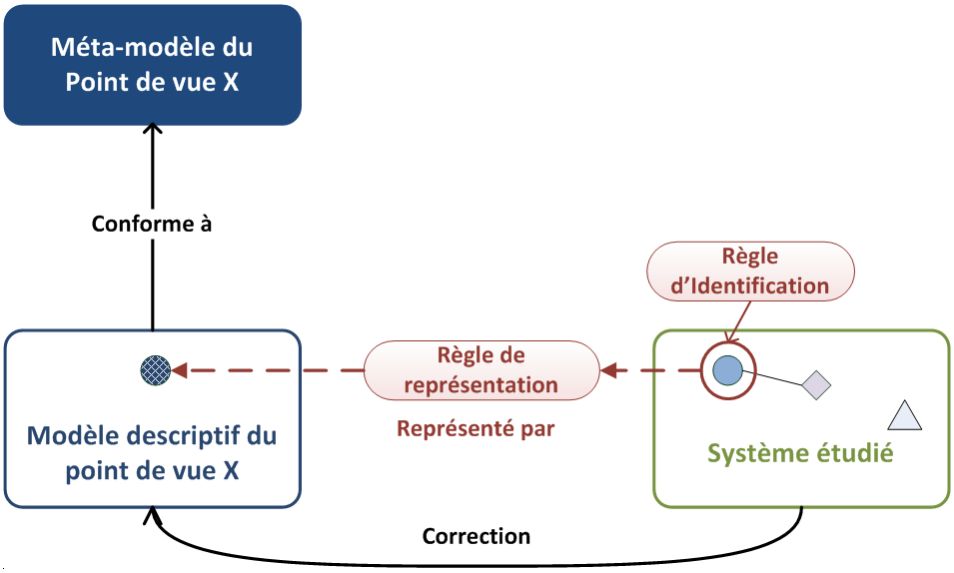


Figure 19 Correction

Les *règles d'identification* permettent de reconnaitre (identifier), au sein du système étudié, les éléments que l'on veut décrire.

Les éléments identifiés deviendront la destination des relations de correction. Les *règles d'identification* définissent le point de vue du système descriptif.

La relation *« représentéPar »* relie des éléments identifiés d'un système étudié à des éléments dans le modèle descriptif.

Une règle de *représentation* représente un élément identifié d'un système étudié dans le modèle descriptif conformément à son méta-modèle. Par conséquent, l'élément du système est - par représentation - *«représentéPar »* un élément du modèle descriptif.

Précédemment le méta-modèle était celui du système étudié ; cependant dans le cas où le méta-modèle n'est pas celui du système étudié il est nécessaire de lui appliquer une transformation. Une transformation de modèle consiste à transformer un modèle source conforme à son méta-modèle en un modèle conforme à un méta-modèle cible. Cette transformation est définie par les règles de transformations établies du méta-modèle source vers le méta-modèle cible (cf. Figure 20).

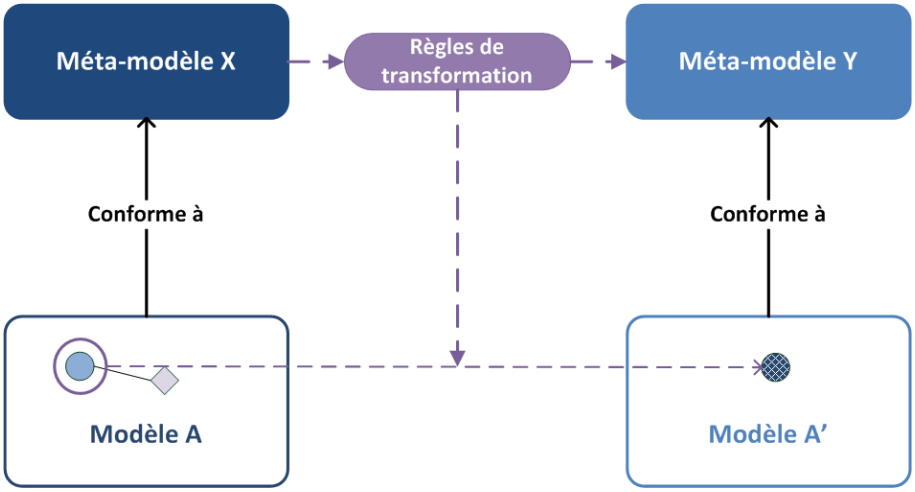


Figure 20 Transformation de modèle

3.4 MODELE A L'EXECUTION

Ces dernières années, ou plus précisément à partir de 2006, nous avons vu émerger le concept de modèle à l'exécution. Cette émergence a été portée par l'engouement à partir de 2001 pour l'informatique autonomique (cf. section 2.2 de ce chapitre) et des systèmes auto-adaptatifs. L'informatique autonomique et l'informatique auto-adaptative se heurtent aux problèmes de la complexité des systèmes à l'exécution [BBF09]. En effet les systèmes que l'on souhaite administrer (cf. [FR07]) :

- Sont souvent distribués et embarqués dans divers équipements ;
- Communiquent en utilisant différents paradigmes d'interaction ;
- S'adaptent dynamiquement aux changements de l'environnement d'exécution.

Or cette complexité des systèmes a creusé l'écart entre le problème que doit résoudre l'application et le problème de l'implantation de l'application. Les tentatives de construction de systèmes qui s'adaptent dynamiquement aux changements de leur environnement ont conduit certains chercheurs à envisager l'utilisation de modèles à l'exécution pour surveiller et gérer l'application. En effet cette approche permet de dissocier la complexité liée aux problématiques à l'exécution du problème que l'on

souhaite résoudre en abstrayant, à l'aide de modèles, le système en cours d'exécution. Ces modèles sont appelés *model@runtime*.

Le but des *models@runtime* est de focaliser le raisonnement de l'adaptation du logiciel sur des aspects du système en cours d'exécution. De ce fait il est impératif que les modèles et le système représenté soient liés causalement (cf. définition de causalité ci-dessus). Il faut que les modèles fournissent les informations exactes et à jour du système pour pouvoir prendre les décisions d'adaptation ; par conséquent si le modèle est lié causalement au système alors l'adaptation au niveau modèle entrainera l'adaptation au niveau système.

Cependant les différents travaux actuels ne réduisent pas les *models@runtime* uniquement à des modèles réflexifs du système. Prenons par exemple les travaux de [MBJFS09] : dans cet article, les auteurs basent leur approche d'adaptation dynamique d'application sur quatre types de modèle et cinq composants.

Les types de modèles sont :

- Les modèles de **ligne de produit de logiciel dynamique (DSPL)** : ils décrivent la variabilité autorisée du système.
- Le modèle du **contexte** : qui spécifie les caractéristiques du système pertinentes pour l'adaptation. Ces caractéristiques peuvent par exemple être fournies par des senseurs.
- Le modèle de **raisonnement** : qui décrit les caractéristiques de résolution du DSPL en fonction du contexte.
- Le modèle d'**architecture** : qui décrit les architectures basées composant.

Les auteurs stipulent que le formalisme des différents modèles n'est pas imposé dans l'approche mais dépend des composants qui utilisent les différents modèles. La Figure 21 schématise les interactions entre les modèles et les composants dans l'approche de [MBJFS09]

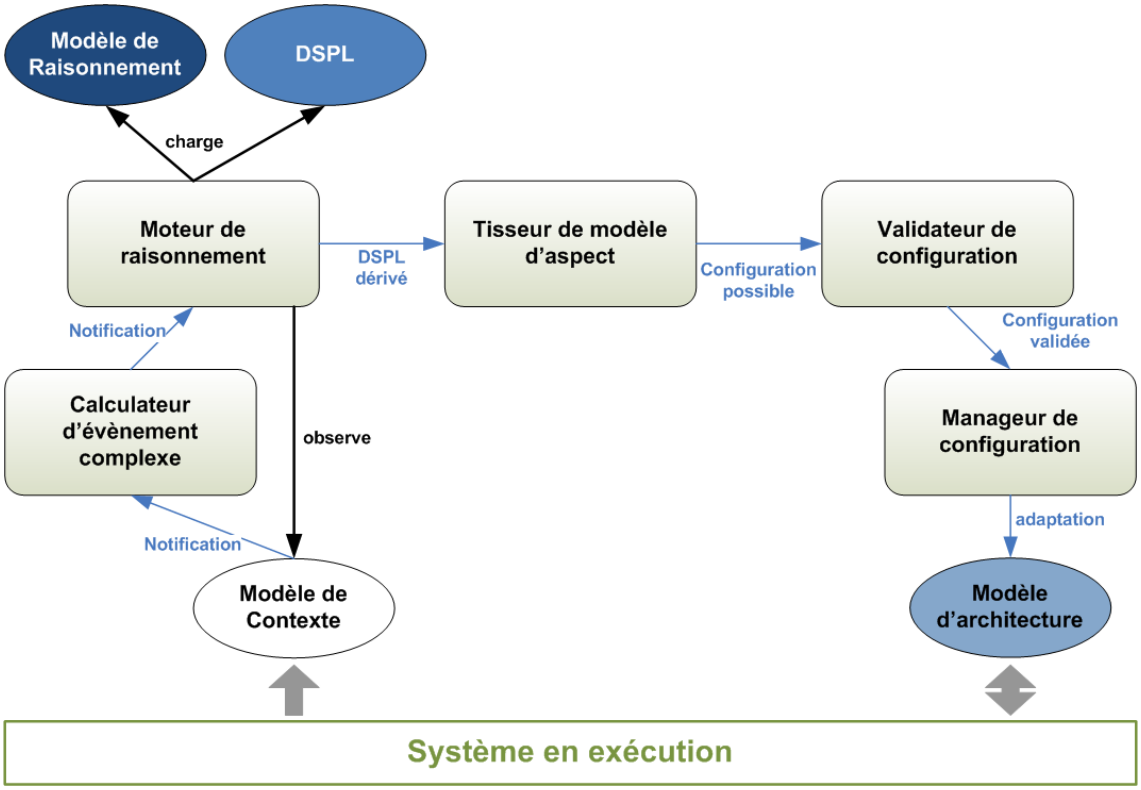


Figure 21 Schéma de l'approche proposée par [MBJFS09]

Ces composants sont :

- Le calculateur d'évènement complexe : ce composant observe l'exécution grâce aux sondes intégrées dans le système et génère les évènements pertinent.
- Le moteur de raisonnement basé sur le but : lorsque le modèle de contexte est mis à jour, ce composant calcule un DSPL dérivé qui contient les caractéristiques obligatoires et une sélection de caractéristiques variables, adaptées au contexte actuel.
- Le tisseur de modèle d'aspect : ce composant reçoit les DSPL dérivés du moteur de raisonnement. Il calcule, à partir des différentes caractéristiques définies dans le DSPL, une configuration possible du système.
- Le validateur de configuration : ce composant récupère la configuration produite par le tisseur de modèle d'aspect et vérifie d'une part si elle est réalisable et d'autre part si elle ne viole pas des contraintes définies par l'utilisateur. Si la configuration est valide alors elle est transmise au manageur de configuration.
- Le manageur de configuration : ce composant est en charge des actions à entreprendre sur le système pour effectuer l'adaptation vers la nouvelle configuration. Pour cela il se base sur le modèle d'architecture qui représente le système en cours d'exécution. Cette adaptation se fait en termes d'ajout, de suppression et de de connexion de composant.

Nous voyons à travers cet exemple qu'il existe plusieurs types de modèles ayant des rôles bien distincts : certains servent à refléter le système en exécution tandis que d'autres servent à définir le système souhaité. Différentes approches adaptatives ont placé les modèles réflexifs de l'exécution et les modèles décrivant l'exécution souhaitée sous la même dénomination de *model@runtime*.

Pour lever cette ambiguïté, Vogel, Seibel et Giese ont catégorisé dans [VGS10] les différents types de *model@runtime*. La Figure 22 présente ces différentes classes de modèles.

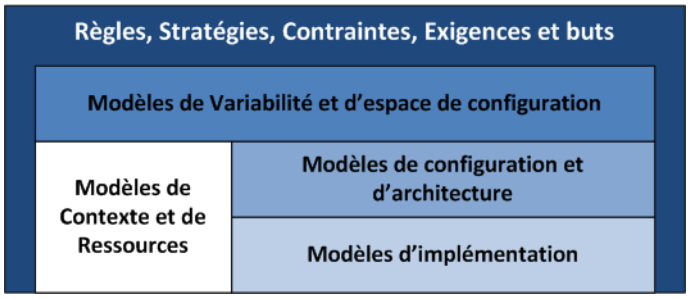


Figure 22 Classification des modèles à l'exécution [VSG10]

- **Les modèles d'implémentation** sont similaires aux modèles utilisés dans le domaine de la réflexion pour représenter et modifier un système s'exécutant grâce à un lien de causalité. Ainsi, ces modèles sont dynamiques, c'est-à-dire qu'ils évoluent en même temps que le système. Ces modèles sont descriptifs et prescriptifs. La granularité est de l'ordre de la classe.
- **Les modèles de configuration et d'architecture** sont à un niveau supérieur par rapport aux modèles d'implémentation. Ces modèles reflètent la configuration/l'architecture courante d'un système. Il est le modèle de base pour la supervision et l'adaptation du système. Ces modèles sont descriptifs et prescriptifs. La granularité est de l'ordre du composant voir du processus.
- **Les modèles de contexte et de ressources** décrivent l'environnement d'un système en cours d'exécution. Ceux-ci englobent toutes les informations pertinentes qui caractérisent une entité, une personne ou un objet pour l'administration souhaitée. Ces modèles sont uniquement descriptifs, ils servent pour le raisonnement de l'adaptation. Par exemple : les caractéristiques matérielles (Système d'exploitation, cadence du CPU, charge du CPU, mémoire totale, mémoire utilisée…) sur lesquelles s'exécutent tout ou partie du système.
- **Les Modèles de Variabilité et d'espace de configuration** définissent les variantes possibles du système, tandis que les modèles de configuration et d'architecture définissent l'état du système

courant. Ce type de modèle définit par intention ou extension l'ensemble des états possibles et autorisés du système.

- **Règles, Stratégies, Contraintes, Exigences et buts** s'appliquent aux modèles des autres catégories et par conséquent, leurs niveaux d'abstraction sont similaires ou supérieurs aux niveaux des modèles visés. Les différents modèles décrivent soit l'état courant du système soit le ou les états souhaités et ou autorisés, alors que les règles, stratégies, contraintes, exigences et les buts définissent : quand, comment et sous quelles conditions on adapte le système courant.

Les catégories présentées par [VGS10] ont des natures et des buts différents et sont complémentaires. Les auteurs soulignent cependant que les catégories utilisées comme le type et le nombre de modèles dépendent de l'approche utilisée (composant, service…) et le domaine métier du système.

3.5 SYNTHESE

L'approche de l'Ingénierie Dirigée par les Modèles (IDM) présente deux principales propriétés : L'abstraction et la séparation des préoccupations. En effet ces deux aspects permettent de séparer la complexité d'un problème (le but d'une application) de la complexité de sa réalisation (la technologie) et de séparer les différentes préoccupations de ce problème. Un modèle est une représentation non exhaustive d'un système mais suffisante et fiable pour un point de vue donné. Le modèle a pour but de se substituer au système original pour ce point de vue.

L'utilisation de l'IDM, que ce soit pour décrire un problème ou pour composer une application a pour but de formaliser des informations pour que celles-ci soient compréhensibles par un humain et par une machine. La présentation de l'IDM dans cette section se focalise avant tout sur la relation entre un système (système étudié), sa représentation (modèle), leur formalisation (méta-modèle) et leur utilisation à l'exécution (causalité…). En effet dans le cadre de cette thèse, l'IDM est utilisée pour abstraire les technologies et fournir une représentation manipulable d'un système en exécution en fonction des préoccupations.

4. PARADIGMES

Dans cette section, nous nous intéresserons aux buts et aux motivations des approches à composant et à service. Nous nous intéresserons à leurs avantages et inconvénients, ainsi qu'à leur concurrence et leur complémentarité.

Nous nous intéresserons dans un premier temps à l'approche à composant, puis à l'approche orientée service. Nous verrons ensuite l'approche qui consiste à utiliser des composants pour implémenter des services. Et finalement nous conclurons par une approche conçut pour des architectures dynamiques basée sur la combinaison des approches à composant et à service.

4.1 APPROCHE A COMPOSANT

L'*approche à Composants*, qui apparaît au milieu des années 90, fait suite à la *Programmation Orientée Objet* (POO ou OOP (anglais)) [Tay98]. Cette approche a été motivée d'une part par le problème récurrent en informatique, qui est la complexité croissante des applications ; et d'autre part par la réduction du temps et du coût de développement et par la spécialisation des acteurs à un contexte dans le cycle de vie de développement [Ba00]. Cette approche promeut la modularisation [Pa72] des applications dans un but de réutilisation [WD01] des éléments logiciels modularisés, nommés : *Composants*. L'idée préconisée par cette approche est de définir les applications en termes d'*Assemblage* de *Composants* [HC01]. Une des particularités des composants est qu'ils suivent le patron « boite-noire » (*Black-Box pattern*). Les applications sont donc définies par les interfaces fonctionnelles des composants indépendamment de leur implémentation. Une conséquence directe est que cette approche permet de dissocier totalement les acteurs de développement des composants de leurs assemblages.

Dans la suite de cette section, nous détaillerons le concept et les caractéristiques des Composants, puis la façon d'assembler des applications. Nous nous intéresserons ensuite au concept d'extensibilités et au comportement des applications à l'exécution. Avant de conclure nous exhiberons des systèmes représentatifs de l'approche à Composant que sont EJB [EJB], CCM [CCMS06] et Fractal [Fra04].

4.1.1 Composant: Classe, Instance ou unité de déploiement ?

S'il n'y a pas de réel consensus sur la définition du *Composant*, il y a néanmoins une définition largement citée et reprise qui est celle de C. Szyperski :

> *"A software component is a unit of composition which contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*
>
> Extrait de [Szy02]

Cette définition est généralement reprise car elle contient les concepts caractéristiques des composants qui sont l'auto-description et la réutilisabilité. En effet, les composants définissent explicitement leurs interfaces fonctionnelles et leurs dépendances (auto-description/introspection), par conséquent il est possible de les composer à partir de ces seules informations ; la connaissance de l'implémentation peut donc être masquée.

Cependant cette définition porte sur différentes matérialisation du concept de Composant :

- Niveau implémentation : la phrase : *"A software component is a unit of composition which contractually specified […]"* définit un composant comme étant une classe/type d'implémentation auto-descriptif

- Niveau unité de déploiement : *"[…] A software component can be deployed […]"* signifie qu'un composant est une unité de déploiement.
- Niveau instance : *"[…] is subject to composition by third parties."* signifie, qu'après être déployé, donc à l'exécution, les composants peuvent être utilisés pour être composé dans des applications à l'exécution donc au niveau instance.

Cette ambiguïté se retrouve très généralement dans la littérature et les plates-formes actuelles. Par exemple, les personnes travaillant au niveau de l'environnement de développement sur les composants verront la notion de composant comme une implémentation, alors que ceux travaillant sur l'assemblage en vue de l'exécution les verront comme des instances.

Prenons par exemple le méta-modèle générique défini par MOF [MOF09] défini dans [TIB05] :

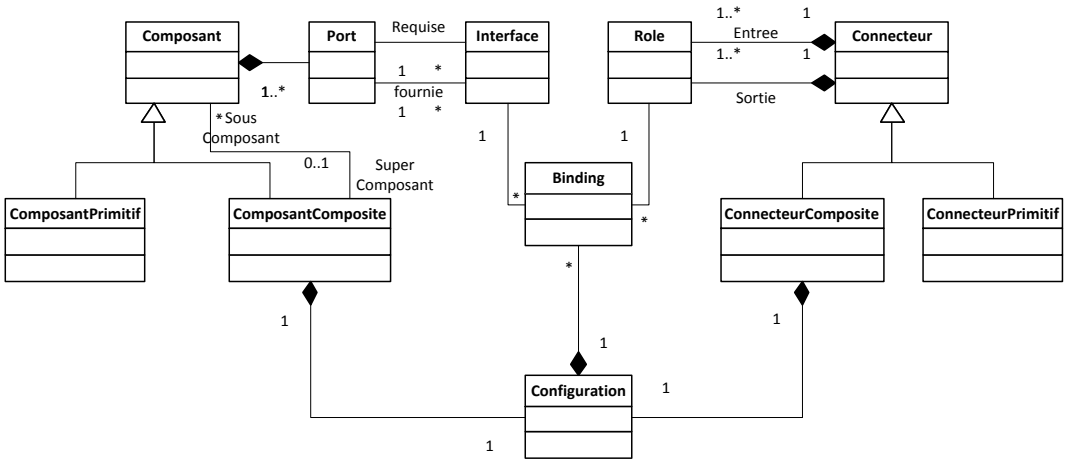


Figure 23 Méta-modèle Composant de l'OMG par [TIB05] (corrigé)

Ce méta-modèle définit l'infrastructure d'exécution des composants et donc ces composants sont définis en termes d'instances et non d'implémentations.

Cette confusion se retrouve dans les ADLs (*Architecture Description Language*) au moment de la conception, car à l'exécution il s'agit bien d'instances. Cette confusion exhibe le problème de la continuité de la *Connaissance*. En effet, la *Connaissance* du composant lors de son développement (implémentation) est différente du niveau de l'assemblage d'un composite (déclaration d'instance) et de l'exécution (instance). Lorsqu'un composant est développé, sa représentation mentale est celle d'une fabrique/classe ; lors de l'assemblage, on la référence comme une déclaration d'instance et finalement à l'exécution on a l'instance et implicitement la fabrique.

Le problème ci-dessus a été pris en compte dans le projet OW2 : MIND[5] [MIND09], qui est une implémentation du modèle à Composant Fractal. Le méta-modèle (cf. Figure 24) obtenu à partir de l'ADL [MIND09] dissocie bien le niveau implémentation et le niveau instance. Cependant, bien que le niveau reste le même entre le développement et l'exécution, son utilisation est différente : à l'exécution le Composant (implémentation) est une fabrique d'instance alors qu'au développement c'est une « classe »/ « type ».

[5] http://mind.ow2.org/

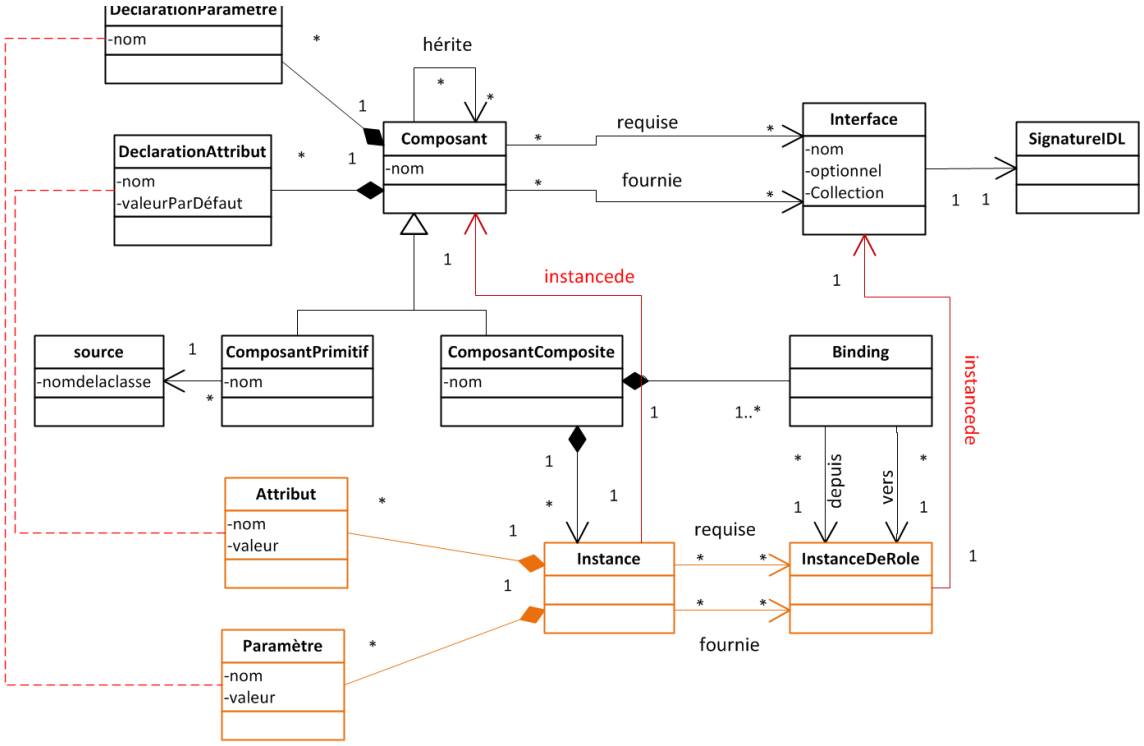


Figure 24 Méta-modèle simplifié de Fractal dans MIND (à partir de l'ADL)

4.1.2 Les applications dans l'approche à composant : Assemblages

Dans l'approche à Composant, l'assemblage des applications se fait par composition structurelle, c'est-à-dire que les éléments constituant l'application sont liés entre eux d'une interface requise à une interface fournie (notion de dépendance).

Assemblage : ADL déclaratif vs. langage de programmation

Généralement, cet assemblage (composition structurelle) se fait au travers d'un ADL (*Architecture Description Language*). Les ADL sont des langages permettant la définition de modèles d'application ; par exemple, la Figure 24 est le méta-modèle de l'ADL fourni dans le projet OW2 : MIND. Cependant dans certaines approches à Composant, comme EJB 3.0 [EJB3.0], les composants sont assemblés par des « clients » : des classes écrites selon le langage de programmation Java, qui instancient les composants et qui les lient de manière programmatique.

Dans le cas des ADL déclaratifs, l'application obtenue est **définie par extension** ; c'est-à-dire que l'ensemble des éléments (implémentation, instance, binding…) qui la composent est connu statiquement au développement. Le modèle d'application des Composants ne traduit pas le but de l'application, mais l'architecture explicite de l'application.

Dans le cas des langages de programmation, la définition et la construction de l'application dépend du code. La définition peut donc être faite par extension comme par intention, et l'architecture peut être figé ou dynamique.

Construction de l'assemblage

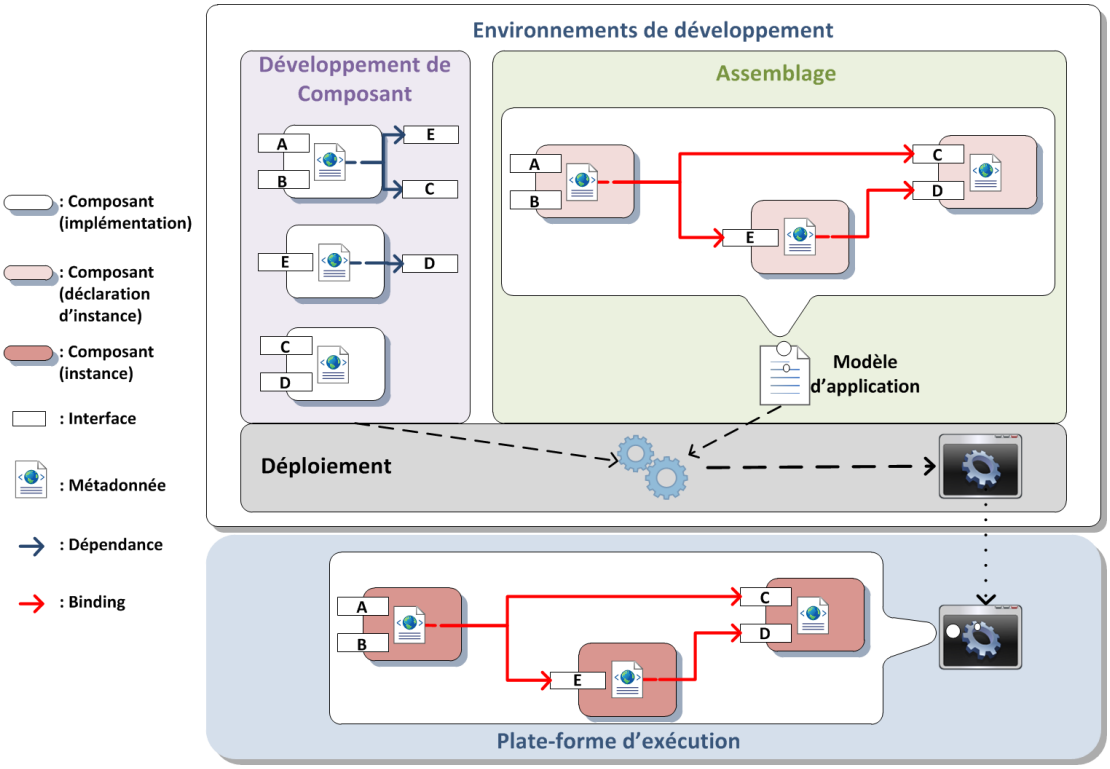


Figure 25 Construction de l'assemblage au développement

Cependant il faut bien voir que le modèle de l'application n'est pas nécessairement interprété à l'exécution, au contraire il est généralement transformé statiquement en binaire avant le déploiement [MIND09] (Figure 25).

Un des principaux avantages du *Composant* est l'introspection qui permet à l'exécution de retrouver l'architecture de l'application. Cependant l'introspection ne permet pas nécessairement de retrouver la *Connaissance* de l'application au développement (cf. partie 4.1.4).

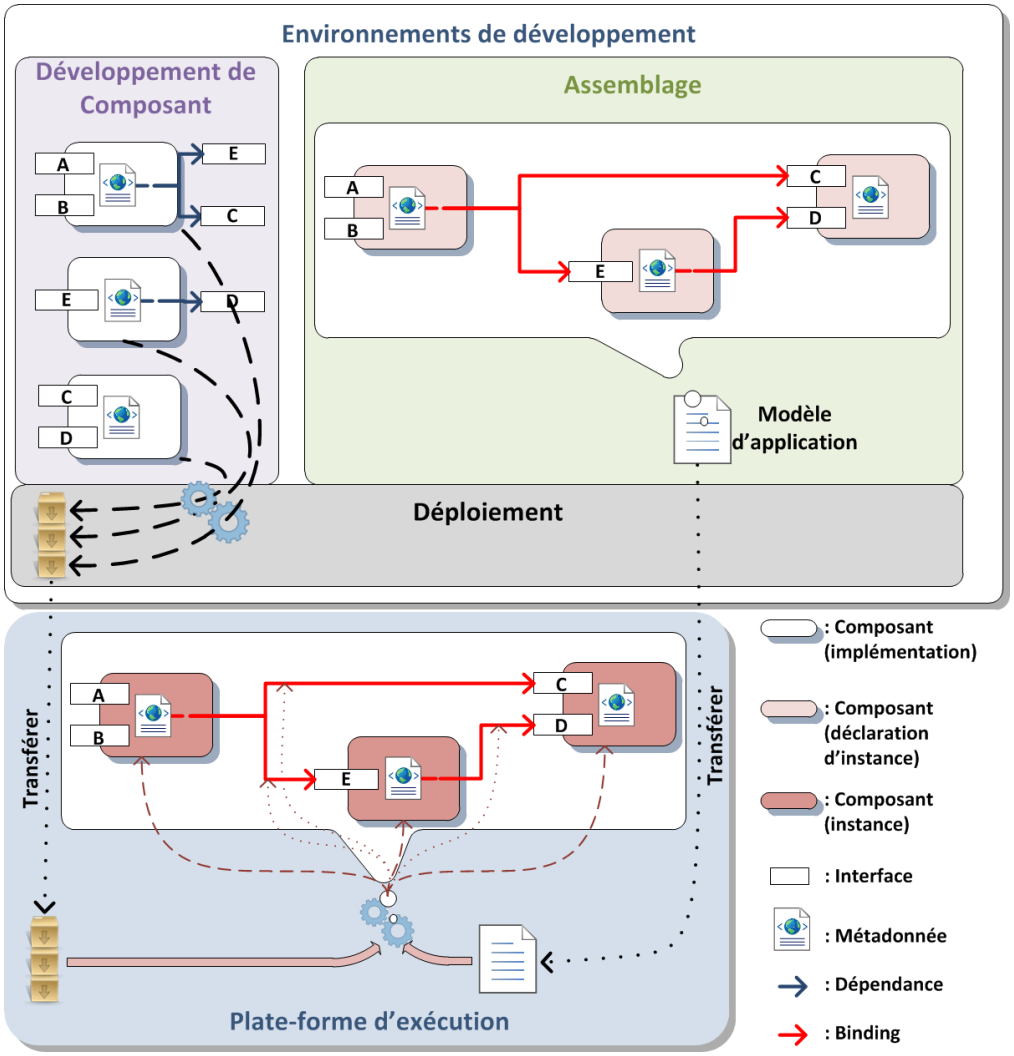


Figure 26 Construction de l'assemblage à l'exécution

Les composants peuvent être considérés comme des classes/types qui sont configurés et instanciés dans différentes applications tierces ; les composants peuvent être utilisés comme des fabriques à l'exécution. Par conséquent, il est possible de soustraire aux « clients » la gestion du cycle de vie de l'application, qui est alors pris en charge par la plate-forme. Cette délégation du cycle de vie à la plate-forme se fait via un conteneur (cf. Figure 27). Un conteneur est un canevas permettant d'administrer un composant. Ce canevas contient d'un côté le code fonctionnel du composant et de l'autre l'ensemble des opérations d'administration du composant. Par conséquent, la plate-forme peut gérer le cycle de vie du composant qui peut alors être plus complexe que la simple création et destruction d'instance. Le modèle de l'application est donc transféré à l'exécution avec ses composants (sous forme d'unités de déploiement). La plate-forme d'exécution a à la fois la *Connaissance* de l'application et la *Connaissance* de l'exécution ce qui permet de charger/instancier l'application (en instanciant les instances de composant à partir des fabriques avec la configuration définie dans le modèle d'application) (cf. Figure 26).

4.1.3 Extensibilité du contexte des composants : Conteneur

Une application n'est pas définie qu'en termes fonctionnels. D'autres préoccupations telles que la sécurité ou les aspects transactionnels sont nécessaires à la bonne exécution de l'application. Les scientifiques travaillant sur des approches à Composants [Szy02] [Fra04], ont proposé une approche se

basant sur la séparation des préoccupations [HL95] [TOHS99]. Les aspects non-fonctionnels des composants sont séparés du code fonctionnel. L'une des approches les plus courantes consiste à les injecter dans le conteneur du composant (cf. Figure 27).

Un bon nombre d'aspects non-fonctionnels ont été identifiés (persistance, sécurité, transaction…) et des solutions standardisées sont disponibles, par exemple la persistance des données est prise en charge par les conteneurs EJB [EJB]. Cependant, certaines approches comme Fractal [Fra04], laissent à l'utilisateur la possibilité de définir ses aspects non-fonctionnels.

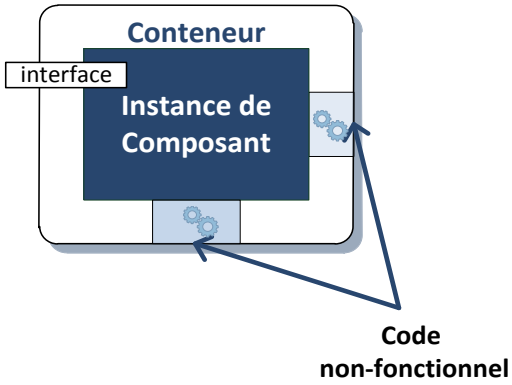


Figure 27 Conteneur et aspects non-fonctionnels

4.1.4 Environnement modulaire, mais statique

Les Composants ont toutes les propriétés requises pour être modularisés : chaque composant expose les ressources requises et fournies. Cependant le fait d'être modulaire ne signifie pas être dynamique. En effet, comme expliqué dans la section 4.1.2 de ce chapitre, les composants sont liés entre eux, or modifier dynamiquement l'architecture signifie rompre une liaison pour en rétablir une autre avec un autre composant , ce qui est rarement autorisé par les ADL qui définissent statiquement une architecture de l'application. Il est à noter toutefois que certaines approches à Composant [Fra04] [DCD06] offrent la possibilité soit de définir des composants abstraits, soit de définir une liaison (*binding*) en fonction d'une interface, qui délègue alors à la phase d'exécution la définition de la liaison (*auto-wire*). Cette possibilité de s'auto-lier lors de l'exécution permet juste une liaison tardive, mais ne peut être en aucun cas assimilé à un comportement dynamique d'une architecture tel que la substitution à l'exécution.

Remarquons de plus que la plupart des approches à composant suivent l'approche décrite dans la Figure 25. Dans ces approches l'environnement d'exécution ne possède pas la *connaissance* de l'application nécessaire pour automatiser totalement ou partiellement les mises à jour ou autres modifications de l'application.

En résumé, les composants sont par nature modulaires, ce qui permet d'améliorer leur réutilisabilité et de faciliter la composition d'application. Cependant cette réutilisabilité et cette facilité ne sont valables qu'à la phase de développement, voire parfois à la phase de déploiement.

4.1.5 Technologies à Composants

Il existe de nombreuses approches à Composant aussi bien dans le monde Académique que dans le monde industriel ; comme déjà cités : les EJB, Fractal, mais aussi COM [Box97], CCM [CCMS06] ou les JavaBeans [JBS97]. Ces approches sont détaillées et comparées dans [CerT04]. Nous allons brièvement décrire les approches proposées par les EJB et Fractal, car elles nous semblent représentatives des approches à Composant courantes.

EJB

L'approche à Composants EJB [EJB] a été conçue pour les architectures en niveaux (*tiers*) [RAJM01]. Il faut remarquer deux points : le premier est que ce genre d'architecture est conçu pour une utilisation multi-utilisateurs du serveur d'application via le niveau présentation. Les instances propres aux utilisateurs sont isolées, car cette approche possède plusieurs politiques d'instanciation : sessionelle, sans état, avec état et entité. Deuxième point : ce type d'architecture est utilisé pour des applications complexes, comme le e-commerce, nécessitant plusieurs caractéristiques non-fonctionnelles telles que la transaction, la sécurité, la distribution et la persistance, que Sun à standardisé pour les plates-formes EJB.

Fractal

L'approche proposée par Fractal est de fournir une plate-forme de composition [BCS02], [BCS04] ne dépendant pas d'un domaine d'application particulier, à contrario des EJB. Il existe de nombreuses variantes de l'approche Fractal telle que Julia, Think ou Mind. En effet, bien que l'approche fournisse un ensemble d'interfaces de programmation pour la réalisation d'aspects tels que la définition de classes de composants, des fabriques, la reconfiguration dynamique des instances, la composition hiérarchique et l'introspection ; elle ne spécifie pas de cycle de vie ni de plate-forme d'exécution, car il dépend souvent du domaine de l'application.

Comparaison

Table 1 Comparatif de technologies à Composant [CerT04]

	EJB	Fractal
Composant (implémentation)	Une seule interface fournie, pas d'interfaces requises, pas de propriétés de configuration.	Vue externe composée de plusieurs interfaces fournies ou requises, obligatoires ou optionnelles, simples ou multiples.
Composant (Instance)	Quatre catégories : session avec état, session sans état, entité, message. Utilisation systématique de fabriques.	Promeut l'utilisation systématique de fabriques. Une seule politique de création (instances multiples).
Unité de déploiement	Fichier Jar pour les composants, EAR pour l'application. Descripteur de déploiement.	Ne définit pas la manière de conditionner un composant.
Assemblage	Impérative à travers le langage de programmation Java	Déclarative avec l'ADL *FractalADL*
Environnement d'exécution	Runtime EJB + conteneurs	Contrôleurs

4.1.6 Synthèse

Comme le résume [MN97], l'approche à *Composant* adresse les problématiques suivantes :

- Rapidité de la mise sur le marché (*Fast time-to-market*) : la réutilisabilité permet rapidement de développer et de mettre une application sur le marché et à moindre coût.
- Fiabilité : Les composants réutilisés sont plus fiables car ils sont, d'une part, déjà testés dans d'autres contextes d'application et, d'autre part, parce qu'ils sont réutilisés : ils sont plus testés.
- Division du travail : des composants avec des interfaces bien définies sont des unités naturelles pour la distribution aux équipes de développement.

- Variabilité : des familles d'applications peuvent être développées en utilisant une base logicielle. Les composants logiciels prennent en charge la variabilité au travers du paramétrage des composants. Les paramètres représentent les fonctionnalités qui doivent être fournies par le client du composant.

Selon l'article [MN97], l'approche à composant, dans le cas où les composants sont bien définis, permet aussi d'adresser les problématiques de distribution, d'hétérogénéité et d'adaptabilité (en tant que flexibilité de l'application) qui prend comme exemple CORBA. Il est intéressant de noter que ces trois problématiques sont à l'origine de l'approche orientée service ; où CORBA est souvent cité comme une des premières technologies à service.

Cependant dans la plupart des modèles à composant, les applications sont liées entre elles au développement ou au plus tard au déploiement. Les propriétés définies par [MN97] sont souvent théoriques. Les besoins de flexibilité, de distribution et de prise en charge de l'hétérogénéité sont à l'origine du paradigme à service. Nous verrons dans les sections précédentes que l'approche à composant et celle à service sont des paradigmes complémentaires.

4.2 APPROCHE ORIENTEE SERVICE

L'apparition de l'approche orientée service (*Service-Oriented Computing* ou SOC) a changé le contexte d'exécution des applications du monde industriel. En effet, peu avant l'an 2000, l'évolution des infrastructures des réseaux en termes de fiabilité et de performance a offert aux entreprises la possibilité de décentraliser et d'interconnecter (B2B[6] [AFHAl99]) leurs applications. Cependant l'approche à composant restreint la flexibilité requise pour ce genre d'application. Par rapport à l'approche à composant, l'approche orientée service offre une plus grande flexibilité et réutilisabilité. En effet, cette approche propose un style architectural permettant de composer rapidement et facilement des applications à partir d'éléments distribués et hétérogènes nommés *services* [PTDLK06]. Cette approche a été plébiscitée au travers d'une de ses implantations : les services Web, qui permettent de définir une application composée de services distribués et hétérogènes.

4.2.1 Définition

L'approche orientée services [Pap03] est basée sur l'idée qu'une application peut être réalisée par composition des services logiciels mis à disposition par des fournisseurs divers ([BLAl00] [TBB03] appellent ce modèle "*service-based model of software*"). Comme pour l'approche à Composant, il n'y a pas de réel consensus sur la définition du concept de *Service*. Cependant, les trois définitions suivantes permettent de recouvrir la majorité des courants du paradigme à Service.

> *"A service is a software resource (discoverable) with an externalized service description. This service description is available for searching, binding, and invocation by a service consumer. Services should be ideally be governed by declarative policies and thus support a dynamically reconfigurable architectural style."*
>
> Extrait de [Ars04]

Dans cette définition, A. Arsanjani recentre la définition de service sur le patron d'interaction promut par cette approche. C'est-à-dire qu'un service est découvrable via une description. Cette description de service est utilisée :

- pour la recherche/sélection ;
- pour l'établissement d'une liaison entre le consommateur et le fournisseur de service ;
- finalement pour l'invocation du consommateur vers le fournisseur.

[6] *Business to Business*

> *"A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. A service is accessed by means of a service interface where the interface comprises the specifics of how to access the underlying capabilities. There are no constraints on what constitutes the underlying capability or how access is implemented by the service provider. A service is opaque in that its implementation is typically hidden from the service consumer except for (1) the information and behavior models exposed through the service interface and (2) the information required by service consumers to determine whether a given service is appropriate for their needs."*
>
> Extrait de [OASI06]

Le consortium OASIS[7] définit un service comme étant un mécanisme permettant d'accéder à une ou plusieurs fonctionnalités. Un service est accessible au travers d'une interface conforme à un ensemble de contraintes et de politiques d'accès. Un service est une boîte noire où seules les informations nécessaires à l'utilisateur pour la sélection et l'invocation sont disponibles. Il faut cependant remarquer qu'il est difficile de discerner une information utile d'une information inutile pour le choix d'un service.

> *"In an SOA, software resources are packaged as "services", which are well defined, self-contained modules that provide standard business functionality and are independent of the state or context of other services.[…] A service in SOA is an exposed piece of functionality with three essential properties. Firstly, an SOA-based service is self-contained, i.e., the service maintains its own state. Secondly, services are platform independent, implying that the interface contract to the service is limited to platform independent assertions. Lastly, the SOA assumes that services can be dynamically located, invoked and (re-)combined."*
>
> Extrait de [PH07]

Dans cette définition, M.P. Papazoglou présente les Services comme étant des entités logicielles qui sont :

- auto-suffisantes pour fournir ses fonctionnalités ;
- indépendantes des autres services et de la plate-forme d'exécution ;
- décrites par des contrats d'interface.

De plus, il considère que les technologies SOA (*Service-Oriented Architecture* ; voir section 4.2.3 de ce chapitre) doivent assumer la découverte dynamique, l'invocation et la reconfiguration des applications orientées services.

Ces trois définitions portent et insistent sur trois aspects de l'approche orientée service : le patron d'interaction [Ars04], l'interopérabilité via une description abstraite [OASI06] et « l'atomicité » et l'indépendance pour la définition des applications [PH07].

4.2.2 Style architectural

Le patron d'interaction que promeut l'approche orientée service est un des principaux apports de cette approche. Il faut cependant remarquer que ce patron d'interaction n'est pas une réelle innovation

[7] *Advancing Open Standards for the Information Society* : http://www.oasis-open.org/

en soi. En effet, il existait déjà, dans les applications distribuées, un niveau d'indirection entre les clients et les serveurs, où les clients passent par le biais d'un intermédiaire qui traduit le nom logique du serveur cible par l'adresse physique (cf. DNS, les courtiers ODP [IRB94]). Cependant l'approche orientée service pousse le raisonnement plus loin : on ne recherche plus un serveur/élément logiciel par rapport à son nom mais par rapport aux fonctionnalités qu'il fournit. En d'autres termes on recherche une fonctionnalité (un service) et non un fournisseur.

Le patron d'interaction du SOC est basé sur trois acteurs :

- Les ***fournisseurs de services*** qui offrent des services
- Les ***consommateurs de services*** qui requièrent et utilisent des services offerts par des fournisseurs
- Un ***registre de service*** permettant aux fournisseurs de publier leurs services et aux consommateurs de découvrir et de sélectionner les services qu'ils veulent utiliser.

Dans ce patron d'interaction, le fournisseur de service publie sa spécification auprès d'un courtier/registre de service. Lorsqu'un Consommateur de service requiert un service, celui-ci recherche, auprès d'un ou plusieurs registres de service, un service conforme à ses besoins ; une fois le service sélectionné, le consommateur interagit avec le fournisseur de ce dernier (cf. Figure 28). Notons que seule la spécification du service est partagée entre les acteurs.

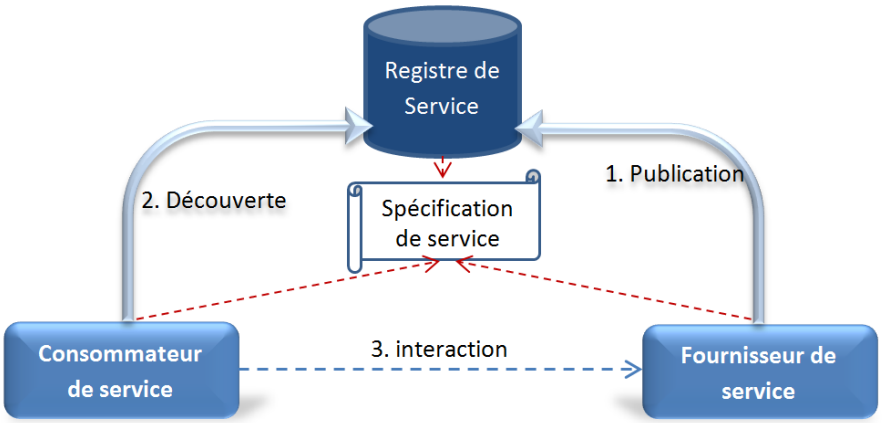


Figure 28 Patron d'interaction du SOC

Grâce à ce patron, les applications orientées service possèdent les caractéristiques suivantes :

- Un **couplage faible** (*Loose Coupling*) : comme la seule information partagée entre les fournisseurs et les consommateurs est la spécification de service, le couplage entre ces deux entités est faible.
- Des **liaisons retardées** (*late binding*) : la liaison entre les fournisseurs et les consommateurs est établie seulement lorsque le consommateur recherche un service et que ce dernier est trouvé.
- Un **chargement paresseux** (*lazy loading*) : une des conséquences de la liaison retardée est de permettre de charger l'implémentation et l'instance du service seulement au dernier moment, c'est-à-dire lors de l'invocation du service.
- Un **masquage de l'hétérogénéité** (*technology neutral*) : grâce au couplage faible, plus précisément grâce à la spécification : un consommateur n'a pas à connaitre les détails d'implémentation du fournisseur de service.
- Un **masquage de la distribution** (*location transparency*) : les consommateurs n'ont pas à connaitre à l'avance la localisation des services. Cette information est obtenue au travers du registre de services.

Un point important est que ce patron d'interaction peut avoir lieu à n'importe quel moment dans le cycle de vie du logiciel; c'est-à-dire que les services peuvent être recherchés lors du développement mais également à l'exécution. Nous nous intéresserons plus attentivement à ce cas dans la suite de ce chapitre.

4.2.3 Service : de l'approche à la technologie

Un service est une entité boîte noire qui fournit des fonctionnalités. L'approche ne dit pas comment implanter un service, elle dit seulement qu'il doit spécifier ses fonctionnalités et les informations « pertinentes » pour qu'un consommateur puisse les sélectionner et y accéder en fonction de ses besoins.

Les technologies mettant en œuvre l'approche orientée service sont appelées des ***architectures orientées service*** (*Service Oriented Architecture* ou SOA) [PaGe03]. Un SOA est un ensemble de technologies permettant de mettre en place des applications suivant le paradigme de l'approche orientée service. Un SOA implante les primitives du SOC en utilisant diverses technologies. Par exemple, les services web sont un SOA, alors qu'OSGi (cf. [OSGia]) en est un autre. Ces deux SOA utilisent des technologies très différentes.

Les technologies choisies pour mettre en place un SOA dépendent du domaine métier et des objectifs poursuivis. Par conséquent, la conception d'un service et la définition d'une application par composition dépendent du SOA utilisé. Tous les SOA ont les mêmes **mécanismes de base** qui assurent la publication, la découverte, la composition, la contractualisation et l'invocation des services. Cependant, chaque domaine métier possède des propriétés non-fonctionnelles et des mécanismes propres (cf. Figure 29) tels que l'administration, les transactions, la sécurité... Dans [PTDL07], M. P. Papazoglou hiérarchise ces mécanismes de base dans une pyramide selon trois niveaux :

- Le niveau fondation (base de la pyramide) contient les fonctionnalités nécessaires à l'approche orientée service : spécification, publication, découverte, sélection et liaison.
- Le niveau composition contient les mécanismes qui permettent de composer une application en termes de services et de valider sa conformité.
- Le dernier niveau contient les mécanismes qui permettent d'administrer et de superviser les compositions et leurs services.

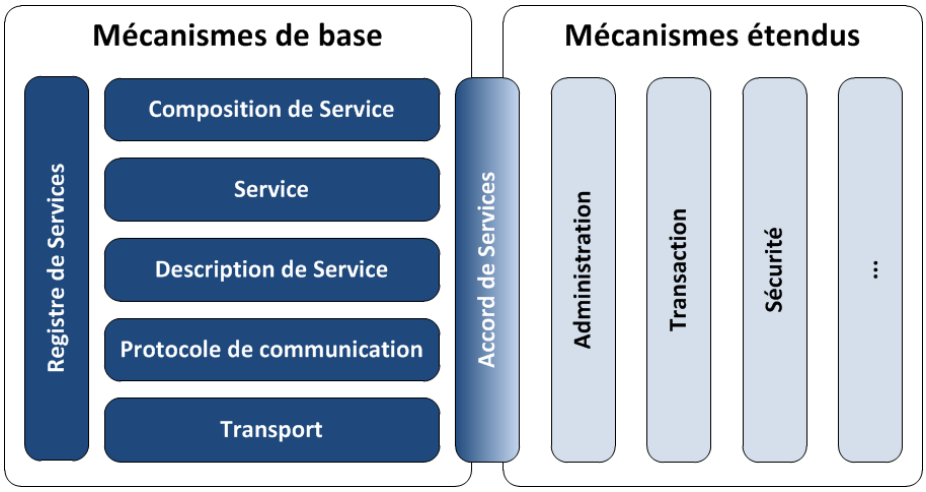


Figure 29 Mécanismes d'un SOA [End04]

De manière transversale à ces niveaux, une architecture orientée service étendue doit prendre en considération la gestion des propriétés non-fonctionnelles, comme la sécurité ou encore la qualité de service. Notons cependant que peu de SOA sont conçus dans l'optique d'être étendus par un domaine métier. Généralement, ces extensions sont standardisées puis incorporées dans le SOA. En d'autres termes bien que les SOA offrent des mécanismes étendus du SOC, cette extensibilité reste en général limitée.

4.2.4 Dynamisme et Sélection

Si on s'en tient à la définition, l'approche orientée service permet d'être hautement dynamique. En effet, le couplage faible permet – via la spécification du service (interface) – de lier un consommateur de service à un service seulement à l'exécution, voir de retarder la sélection du fournisseur et la création de la communication au moment de son utilisation. Cette propriété, nommée **liaison tardive** (*Late Binding*), était déjà utilisée par l'approche à composant (cf. section 4.1.4 de ce chapitre). Cependant, le fait que la communication entre un client et un fournisseur se fasse par sélection, offre un second avantage : la substituabilité. En effet, un service est considéré sans état [OASI06] [PH07], conséquemment il est possible de substituer un fournisseur de service par un autre entre deux invocations.

La substitution peut se faire pour deux raisons :

- Le service utilisé par le consommateur n'est plus disponible, il faut donc qu'il en sélectionne un autre (auto-réparation cf. Figure 12) ;
- Un autre service est disponible offrant de meilleures caractéristiques (par exemple une meilleure qualité de service) (auto-optimisation cf. Figure 12).

La deuxième raison (auto-optimisation) nécessite un mécanisme de notification. Il faut noter que ce mécanisme n'existe pas nécessairement dans tous les SOA.

En résumé, outre la flexibilité apportée par la liaison tardive et le chargement paresseux (*lazy loading*), l'approche orientée service permet théoriquement la définition d'applications dont l'architecture ou la composition change dynamiquement à l'exécution. Cependant cela reste souvent théorique (cf. les sections 4.1.1 et 4.3.3 de ce chapitre).

4.2.5 Synthèse

Le but de l'*approche orientée service* est de permettre la construction d'applications à partir d'entités logicielles particulières, les services, tout en assurant un couplage faible. Cette approche n'est pas une technologie ; elle peut être vue comme un style architectural.

Dans cette approche les concepts d'implémentation ou d'instance n'ont pas de sens, car cette approche définit un patron d'interactions permettant de composer des applications à partir de ressources atomiques, partageables et potentiellement distribuées. Dans cette vision, ce qui est important c'est que l'on puisse accéder aux fonctionnalités ; « savoir quelle est son implémentation », « savoir si l'on accède à une instance » sont des concepts non pertinents. En effet ce qui importe pour un utilisateur c'est que le service soit une spécification de fonctionnalités atteignables au travers des points d'invocation.

Par conséquent, pour définir une application orientée service il faut une technologie SOA fournissant les mécanismes de base ainsi que des mécanismes étendus pour le support de propriétés non-fonctionnelles. Il existe de nombreux SOA (CORBA [Corb08], Jini [Jini], service Web, OSGi [OSGi-s], iPOJO [iPOJO-s]), qui peuvent être catégorisés selon deux approches du SOC, qui sont :

- Composant à Service ;
- Composant Orienté Service.

Les sections 4.3et 4.4 de ce chapitre détailleront ces deux approches.

4.3 COMPOSANT A SERVICE

Du point de vue d'un consommateur, un service est un composant logiciel atomique et partageable. Par conséquent, il est relativement aisé de réaliser/composer une application orientée service. Par contre, lors de la phase de développement, les développeurs se concentrent uniquement sur les fonctionnalités du service à implémenter indépendamment de son contexte d'utilisation. Dans le cas des *Web-Services* (WS), l'utilisation de composants (paradigme) pour implémenter des services est actuellement admise et partagée par la communauté des services [Yan03].

Les *Web Services* sont certainement le SOA et l'implémentation des services Web les plus connus et les plus populaires dans le monde industriel et académique. En effet, les *Web Services*, en plus de supporter la distribution et l'hétérogénéité des services, permettent de définir des applications orientées service rapidement et facilement. Cette facilité est due à une séparation quasi hermétique entre la réalisation de l'application et la réalisation de ses services.

4.3.1 Web Services

Les services Web sont une dénomination regroupant l'ensemble des SOA basé sur la distribution sur le Web. Cette dénomination regroupe les technologies comme CORBA, Jini ou encore les *Web Services*. Dans cette section nous nous intéresserons aux Web-Services, c'est-à-dire l'ensemble des technologies des services Web qui implémentent les spécifications WS-*.

Le principal atout de la technologie des Web services est de fournir un cadre d'interopérabilité distribué qui a permis aux industriels d'interconnecter leurs systèmes ([AF01] et [CNW01]) pour un coût réduit. En effet, la technologie des Web services sépare l'utilisation du service de sa réalisation, par conséquent, elle permet aux clients de services de les utiliser sans avoir à connaître les détails techniques de son implémentation [ACKM04]. Les Web Services sont standardisés à la fois par le W3C[8] et le Consortium OASIS. Les Web Services utilisent un ensemble de standards (WS-*) dont les principaux sont résumés dans la Figure 30.

Notons cependant que les registres de services sont rarement utilisés dans le milieu industriel.

Actuellement, il existe très peu de méthodes de développement pour l'approche à service (cf. Chapitre 2 - section 1.1 : SOMA) et elles n'ont pas été validées hors du contexte dans lequel elles ont été définies. Le fait qu'il y ait peu de méthodes spécifiques à l'approche orientée service s'explique, outre son apparition récente, par le fait que l'approche orientée service promeut principalement un patron d'interaction entre les éléments de l'application *lors de l'exécution*. De ce fait, le choix de la méthode de développement d'application n'a que peu ou pas d'incidence sur l'exécution.

[8] *World Wide Web Consortium*. http://www.w3.org/

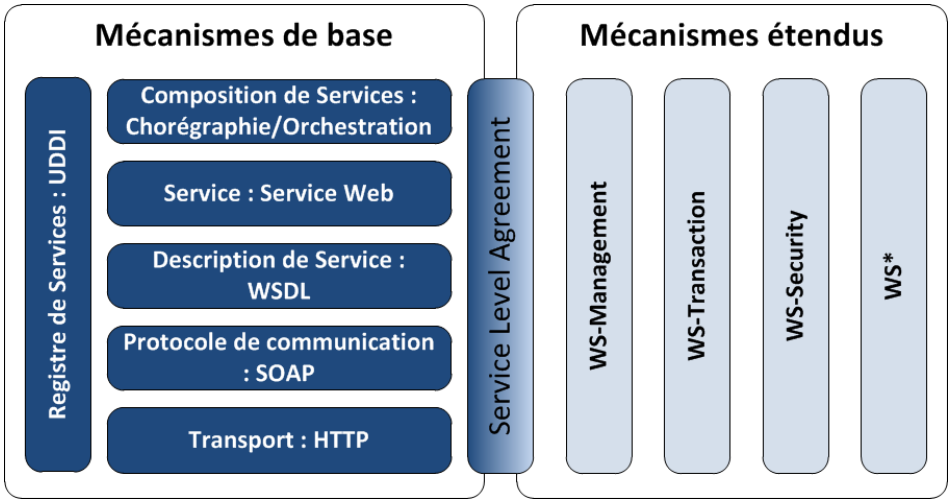


Figure 30 WS : technologies utilisées

Globalement le cycle de vie d'une application est décomposé en deux parties : (1) le cycle de vie du développement et (2) le cycle de vie de l'exécution. Durant le cycle de développement, l'application est définie, implémentée, testée… Suit le cycle de vie de l'exécution : ce cycle est lui-même divisé en deux phases : le déploiement (cf. Chapitre 2 - section 1.2) et l'exécution proprement dite (cf. Chapitre 2 - section 1.3).

Un des aspects importants, particulièrement mis en avant par les approches orientées Web Service, est la séparation entre la réalisation d'une application et la réalisation des services qui la composent.

4.3.2 Réalisation d'un Service par un composant

Comme exposé dans la section 4.2 de ce chapitre, l'approche orientée service définit une façon de réaliser une application à base d'entités logicielles nommées services, mais ne dit en aucune façon comment réaliser ces services. De nos jours, la manière la plus courante de réaliser un service est de promouvoir un composant en tant que service Web. La quasi-totalité des SOA utilisant des *Web Services* ont recours au *Composant à Service.*

> *Un* **Composant à Service** *est un composant promu en tant que service, par l'ajout d'une description et d'une interface propre à une technologie SOA.*

Dans cette approche, certains composants sont promus en tant que services en ajoutant dans ses métadonnées une description des services fournis sous la forme d'une interface (indépendante de l'implémentation) et de propriétés. L'aspect service est uniquement utilisé pour la définition d'application. Dans cette approche les composants ignorent la notion de service ; du point de vue du développeur, le niveau application est non pertinent, seule compte la notion de service.

Notons cependant qu'une application basée sur des composants considérée comme ayant une « bonne » architecture et de « bonnes » performances ne va pas nécessairement faire une « bonne » application orientée service.

Cette façon de faire offre plusieurs avantages :

- Elle permet de réutiliser l'infrastructure des entreprises et les applications déjà existantes ; or un service est un excellent moyen d'exposer une vue extérieure (interface d'invocation distribuée) d'un système, en réutilisant en interne la conception traditionnelle des composants.
- Elle permet de dissocier l'implantation d'une application de ses services, permettant ainsi de définir facilement et rapidement des applications de plus grandes granularités.

- Elle permet, grâce à une interface abstraite, d'interconnecter différentes applications de technologies hétérogènes d'une entreprise (cf. : *Entreprise Service Bus*).

Le principal avantage de cette approche est que le Service est atomique et partageable, et que la distribution n'est pas pertinente. Par conséquent, une application orientée services peut utiliser librement les services déjà utilisés par d'autres applications, en local comme à distance.

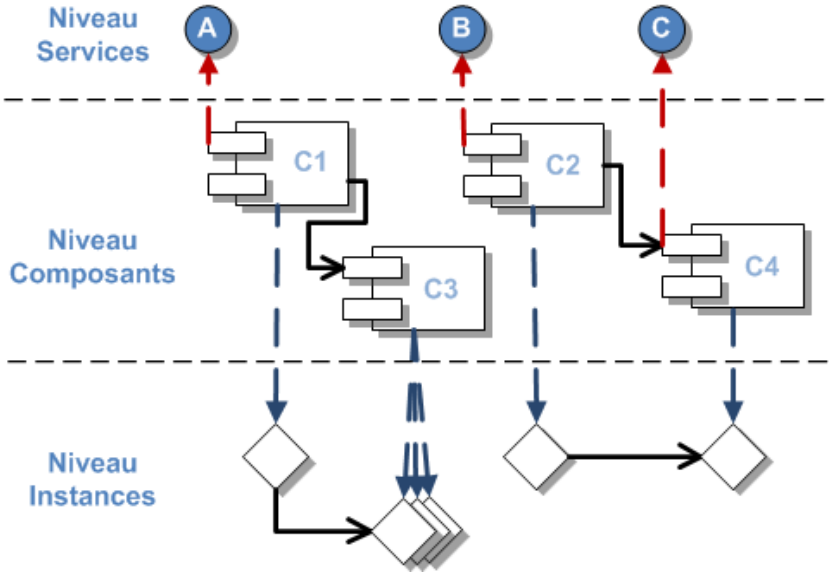


Figure 31 Les trois couches de réutilisations [End04]

Un des aspects importants dans l'approche orientée *Web Service*, est la séparation entre la réalisation d'une application et la réalisation des services qui la composent. C'est-à-dire que du point de vue de l'application, les *Web Services* sont des entités logicielles « boîte noire », où les concepts d'implémentation, d'instance et de dépendance n'ont pas de sens. Inversement, du point de vue du développeur du service, c'est concept d'application qui n'a pas de sens ; car seules comptent les fonctionnalités des services à implémenter. En partant de cette constatation il n'est pas possible de définir une application par composition structurelle (*Structural Composition*), mais seulement par composition comportementale (*Behavioural Composition*)

4.3.3 Réalisation d'une application par Composition

Il existe deux approches principales pour la composition de services Web [Pel03] : l'orchestration et la chorégraphie. Ces deux approches sont des compositions comportementales.

- Le but de l'orchestration est de définir un ensemble d'étapes/processus coordonnés par une entité centrale (moteur d'exécution) qui gère l'invocation des différents services intervenant dans la composition selon la logique définie par le procédé. L'orchestration de services est une vision centralisée de la logique de composition qui se concentre sur l'activité de création de processus utilisant des services web. Dans une orchestration, une application est modélisée comme une succession d'activités où chaque activité correspond à l'invocation d'une opération d'un service web (cf. Figure 32). Exemples d'orchestration : BPEL4WS [ACDAl03] et FOCAS [PE08] [PE09] [PedT09]

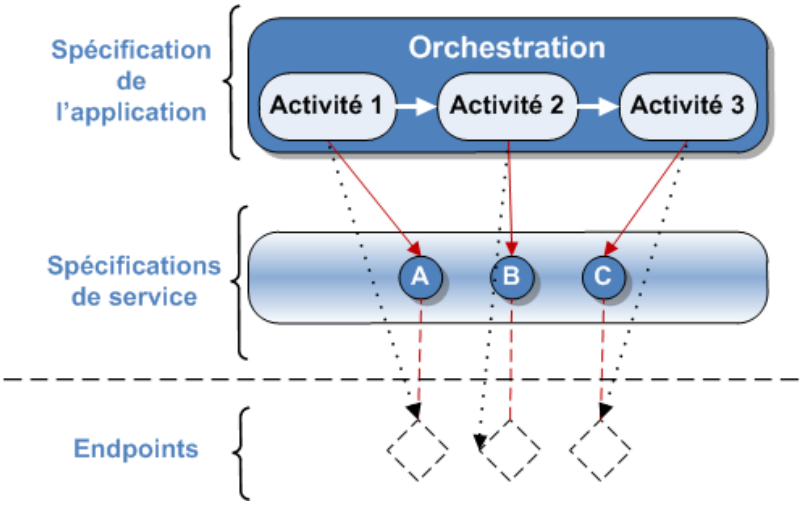


Figure 32 Orchestration

Le but de la chorégraphie de services est de décrire la manière dont un ensemble de services interagissent pour atteindre un but commun. La chorégraphie spécifie les interactions des services participants et les dépendances entre ces interactions. Ces dépendances peuvent être des dépendances de flot de contrôle (une interaction doit être précédée par une autre), des dépendances de données (une interaction nécessite la réception d'un message avant de débuter) etc. Elle ne décrit pas l'implémentation des services participants [BDO05]. En résumé, la chorégraphie se concentre sur les interactions entre différents acteurs (typiquement l'échange public de messages entre des services web). Exemple de chorégraphie : WSCI [AAAI02] Le dynamisme avec les Composants à Service

Il est important de remarquer que le plus souvent, une fois l'application transférée, configurée et activée, « l'architecture » de l'application n'« évolue » pas tant qu'il n'y pas de mise à jour. Potentiellement, des paramètres fonctionnels et/ou non-fonctionnels sont modifiés par un administrateur. Bien que l'application qui en résulte possède les propriétés de *liaison retardée* et de *chargement paresseux*, l'exécution de l'application n'en reste pas moins statique. En effet, bien que le *couplage faible* permette de mettre en place des points de variabilités et donc de substituer dynamiquement un service par un autre, cette possibilité n'est pas ou très peu utilisée. Le monde industriel cherche, à juste titre, à fiabiliser et à sécuriser l'exécution des applications ; or le « dynamisme » tel que la substituabilité est une source d'erreurs car il augmente le nombre de scénarii et d'architectures à tester et est donc moins fiable qu'une architecture dite statique.

Il faut bien voir que la séparation entre l'application et les éléments qui la composent (les services), implique que l'évolution de l'application diffère de l'évolution des services. Dans le contexte des Web Services, la disponibilité d'un service est quasi-permanente ; de plus les Services Web sont souvent utilisés dans une logique de B2B. Dans ce cas si une entreprise paye un abonnement pour utiliser un service, elle n'a pas nécessairement envie d'en sélectionner un autre. De fait en pratique, les entreprises utilisent rarement le mécanisme de sélection préférant assigner à chaque client un Service Web identifié et souvent contractualisé.

En résumé, bien que les mécanismes de notification et de sélection sont spécifiés et implantés, ceux-ci sont rarement utilisés en pratique dans le milieu industriel.

4.3.4 Synthèse

Lors de la phase de développement, les développeurs se concentrent uniquement sur les fonctionnalités du service à implémenter, indépendamment de son contexte d'utilisation. Dans le cas des Web-Services, l'utilisation de composants (paradigme) pour implémenter des services est actuellement admise et partagée par la communauté des services [Yan03]. Certains composants sont

promus en tant que services en ajoutant dans ses métadonnées (WSDL) une description des services fournis sous la forme d'une une interface (indépendante de l'implémentation) et de propriétés. Dans cette approche (cf. 4.3), les entités logicielles sont des composants (paradigme), ils peuvent donc avoir des dépendances envers d'autres composants. L'aspect service est uniquement utilisé pour la définition d'application. Les composants ignorent la notion de service ; du point de vue du développeur, le niveau application est non pertinent, seule compte la notion de service.

Le principal avantage de cette approche est que le Service est atomique et partageable, et que la distribution n'est pas pertinente. Par conséquent, une application orientée service peut utiliser librement les services déjà utilisés par d'autres applications, en local comme en distant.

Cependant, cette séparation quasi-hermétique, lors de l'exécution, entre le niveau applicatif et la plate-forme d'exécution fait que l'application offre une vue « utilisateur » de services qui ignore les détails de mise en œuvre, tandis que les composants ont une vue « fournisseur » qui ignore pour quelle application ils « travaillent ». Par conséquent, une orchestration peut reconfigurer des applications en changeant par exemple un service par un autre ; mais, n'ayant aucune « connaissance » des composants et de leurs dépendances, le moteur d'orchestration ne peut pas déployer de nouveaux services.

En résumé, dans cette approche il y a une séparation entre le niveau applicatif et la plate-forme d'exécution qui fait que leurs cycles de vie sont quasi-hermétiques et que leurs *Connaissances* ne sont pas partagées; ce qui est bien l'intention initiale. Cependant, cette séparation entre la *Connaissance* de l'application et la *Connaissance* de l'exécution empêche la plate-forme d'exécution d'adapter l'application en fonction des besoins. Réciproquement, le niveau applicatif n'est pas en mesure de modifier l'état de la plate-forme d'exécution en fonction de ses besoins. Cela rend extrêmement difficile l'adaptabilité et la dynamicité des applications orientées service (discuté dans [PTDL07], section *Research Challenges*, pp.41), [BBF09] and [FHSAl06].

4.4 COMPOSANT ORIENTE SERVICE

L'approche à composant et l'approche composant à Service possèdent de nombreux avantages comme la réutilisabilité, la flexibilité, la robustesse… mais possèdent une limitation non négligeable pour certains domaines – comme l'informatique ubiquitaire – qui est la staticité de l'architecture de l'application. Dans [CerT04], Humberto Cervantes explique le fait qu'en générale les implémentations de l'approche à composant sont limitées vis à vis de la possibilité de supporter la disponibilité dynamique car :

- Les applications sont assemblées en phase de conception et non à l'exécution.
- Dans les applications composées de façon déclaratives, le but de l'application est défini par extension et n'autorise donc pas les changements à l'exécution.
- Dans le cas d'une composition impérative, il est possible de reconfigurer l'application (création et destruction d'instance, établissement de connexion) mais seulement avec des classes de composants déjà disponibles durant la phase de conception.
- La disparition d'une classe de composant à l'exécution n'est pas une hypothèse de l'approche à composants.

Pour combler cela, H. Cervantes propose d'introduire l'approche orientée service pour la communication entre les composants d'un modèle à composant. Il nomme cette approche : composant orienté service (*Service-Oriented Component*) [CerT04] [CeRi04].

> *"Service-Oriented Component model introduces concepts from service orientation into a component model. The motivation for such a combination emerges from the need to introduce explicit support for dynamic availability into a component model."*
>
> Extrait de [CeRi04]

Il existe deux différences majeures entre les composants à service et les composants orientés service ; la première se situe dans la communication entre les composants, la seconde dans la logique de composition d'application.

Dans cette section, nous allons dans un premier temps nous concentrer sur les concepts et les mécanismes. Ensuite nous étudierons la composition d'application suivant la logique de modèle de composant orienté service ; puis finalement nous nous intéresserons à deux technologies qui suivent cette approche.

4.4.1 Concepts et mécanismes

La caractéristique primordiale des modèles à composant orienté service réside dans l'interaction entre les composants. En effet, dans cette approche les composants ne sont pas liés entre eux directement ; les composants exposent leurs fonctionnalités sous la forme d'un ou plusieurs services et donc utilisent les fonctionnalités des autres composants aux travers des services qu'ils exposent. En résumé, dans les modèles à composant orienté service le *binding* est remplacé par l'approche à service (publication, recherche, sélection, utilisation).

La Figure 33 résume les différentes caractéristiques d'un composant orienté service :

- Interfaces de Service fournies : les fonctionnalités du composant sont exposées via une description, dans le cas de [CerT04], cette description est une interface Java.
- Dépendances de Service requises : les dépendances de fonctionnalité sont déclarées par le composant et prises en charge par le conteneur. La résolution de la dépendance du conteneur se fait suivant l'approche orientée service.

- Interface et dépendance de contrôle : les interfaces de contrôle permettent aux instances de composants de participer aux activités de reconfiguration dynamique.
- Propriétés de configuration : propriétés fonctionnelles du service.
- Propriétés de Service : propriétés exposées avec les interfaces de service
- Exposition et dépendances de ressources : ce sont des références à des ressources exposées qui peuvent être utilisées par d'autres composants. Dans le cas où elles sont requises, celles-ci sont recherchées dans les ressources exposées par les autres composants.

La combinaison de l'approche à composant et celle des services permet aux développeurs de se concentrer sur le code fonctionnel / métier et de déléguer la gestion de la disponibilité dynamique ou du cycle de vie des services (composants) aux conteneurs.

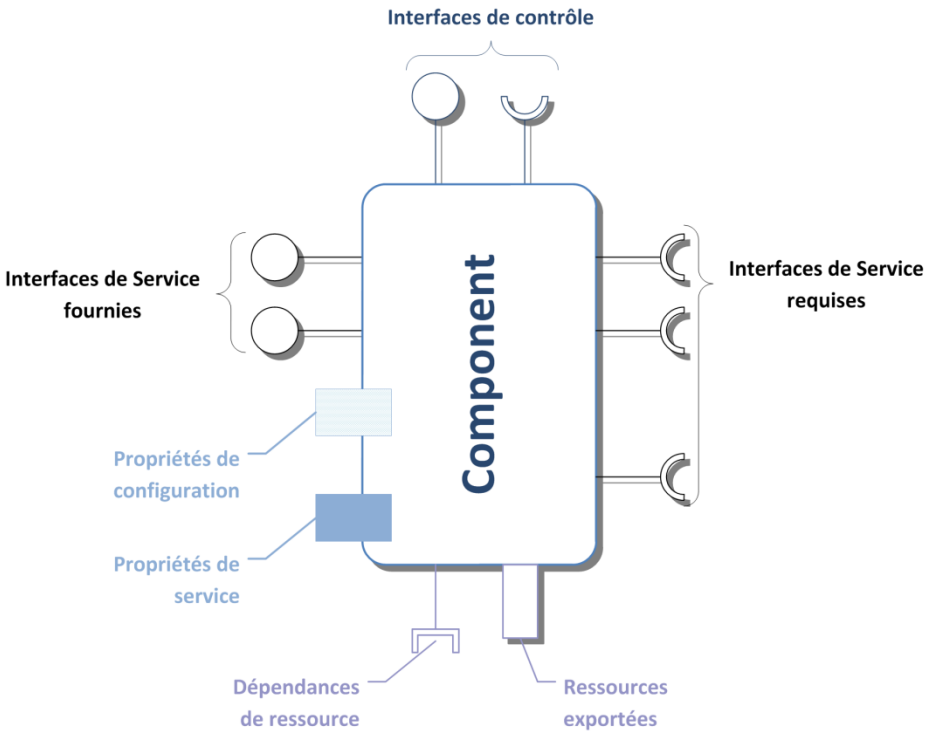


Figure 33 Composant Orienté Service défini par [CerT04]

Dans certaines technologies à service comme OSGi, le code qui gère la disponibilité dynamique des services est assez complexe et est à la charge du développeur. L'utilisation de conteneurs inspirée des composants pour gérer des aspects non fonctionnels, va permettre de gérer, entre autres, la disponibilité dynamique, ce qui va simplifier la tâche des développeurs [DieT10].

4.4.2 Logique de composition d'application

L'approche composant orienté service permet d'une part de concevoir des applications suivant une composition comportementale comme l'orchestration (cf. la section 4.3.3 de ce chapitre), mais permet aussi la composition structurelle (cf. la section 4.1.2 de ce chapitre). Comme l'approche à composant, les applications de composants orientés services peuvent être définies suivant un ADL (cf. SCA à la section 4.4.3 de ce chapitre), ou de manière programmatique.

Dans le cas d'une application définie par un ADL, pour pouvoir gérer la disponibilité dynamique des éléments qui la composent, il faut respecter deux critères :

- Assembler l'application à l'exécution de manière similaire à la Figure 26 ;

- Définir les points de variabilité dans l'application où les éléments sont susceptibles d'avoir une disponibilité dynamique.

Dans le cas d'une application définie de manière programmatique, il est intéressant de remarquer qu'il est possible de composer une application seulement de manière opportuniste. C'est-à-dire de développer un ensemble de composants orientés service où chacun se lie aux autres en fonction de ses dépendances. Si l'ensemble de ces composants sont déployés sur le même environnement d'exécution vierge, alors l'architecture de l'application devrait être conforme à nos attentes et ce sans avoir de modèle d'application, cette façon de faire est celle d'OSGi [OSGia]. Cependant si l'environnement d'exécution est ouvert, il n'est pas possible de prévoir l'architecture de l'application obtenue.

Nous voyons que le modèle composant orienté service autorise deux façons diamétralement opposées (du point de vue du dynamisme) de composition d'application : d'un côté des applications définies statiquement et de l'autre des applications se construisant dynamiquement à l'exécution par opportunisme. Entre ces deux extrêmes, il y a la possibilité de définir des applications dans lesquelles certains « morceaux » de l'architecture sont autorisés à varier au cours de l'exécution.

4.4.3 Systèmes représentatifs

Dans cette section nous allons aborder succinctement les modèles composants orientés service iPOJO et SCA. Ces deux modèles couvrent globalement le spectre des différentes possibilités qu'offre ce modèle. Ces deux modèles seront expliqués plus en détail dans les chapitres 3 et4 de cette thèse. Dans cette section nous nous intéresserons seulement aux concepts (du méta-modèle), au dynamisme et à la séparation des préoccupations non-fonctionnelles ; les parties sur la définition des applications ne seront pas traitées celles-ci ayant déjà été abordées dans [DieT10].

iPOJO

Le modèle composant orienté service iPOJO [EscT08] [EHL07] [iPOJO-s] – pour *injected Plain Old Java Object* – est développé au-dessus de la plate-forme OSGi. Son objectif est de fournir une infrastructure de développement et d'exécution d'applications dynamiques suivant le paradigme à service. Le modèle iPOJO est le résultat de la thèse de C. Escoffier [EscT08] ; il enrichit l'approche proposée par H. Cervantes sur deux points : il utilise l'injection de byte code pour instrumenter le conteneur à la compilation ou à l'exécution, il propose une approche qui permet d'ajouter facilement des aspects non-fonctionnels aux composants sans avoir à modifier le code métier.

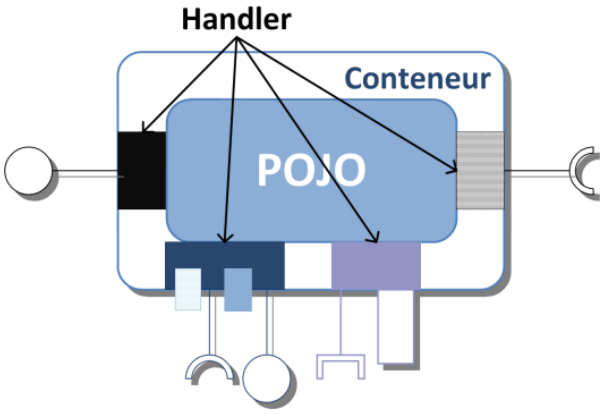


Figure 34 Composant orienté service dans iPOJO

Le modèle iPOJO applique le principe de la séparation de préoccupations à tous les aspects non-fonctionnels en se servant du concept de conteneur. Il y a trois éléments dans un composant iPOJO (cf. Figure 34) : le premier est le code métier qui s'écrit sous forme de POJO (objet java). Le deuxième élément est le conteneur du composant, et le troisième élément est l'ensemble des aspects non-

fonctionnels injectés dans le conteneur. Chaque aspect non-fonctionnel est défini par un *« handler »* qui est lui-même un composant iPOJO. Ce « *handler* » définit la logique de l'aspect non-fonctionnel ; à l'exécution il est instancié et injecté dans le conteneur du composant.

iPOJO prédéfinit un ensemble de *« handlers »* qui correspondent aux différents points proposés dans *Service Binder* de [CerT04] exposés dans la Figure 33. Ces *« handlers »* prédéfinis sont :

- *Provided Service Handler* : équivalent à « Interfaces de Service fournies » et « Propriétés de Service » ;
- *Service Dependency Handler* : équivalent à « Dépendances de Service requis » ;
- *Lifecycle Callback Handler* : équivalent à « Interface et dépendance de contrôle » ;
- *Configuration Handler* : équivalent à « Propriétés de configuration ».
  La gestion des ressources est comme dans le cas de [CerT04] délégué à la plate-forme OSGi.

SCA

L'approche de SCA (*Service Component Architecture*[9]) est issue d'une collaboration OSOA[10] entre différentes entreprises comme IBM, Oracle, BEA Systems…

L'objectif affiché de SCA est de fournir un ensemble de spécifications fournissant un modèle pour la création de composants, et un modèle de programmation permettant de construire des applications logicielles selon l'approche orientée service. Un des avantages majeurs de SCA est que sa spécification supporte plusieurs langages comme Java, C++, C#... et plusieurs technologies comme BPEL [ACDAl03] pour la composition. En d'autres termes SCA cache l'hétérogénéité des langages et technologies utilisés par un modèle abstrait.

La spécification SCA identifie plusieurs concepts :

- Service : un service représente une interface d'invocation d'une implémentation (équivalent à « Interfaces de Service fournies »).
- Implémentation (*Implementation*): une implémentation de composant est l'implémentation concrète de la logique métier qui fournit ou permet de référencer des services.
- Composant (*Component*) : un composant est une instance configurée d'implémentation. Un composant fournit et requiert des services. Il peut y avoir plusieurs composants pour une même implémentation, cependant la configuration est propre à chaque composant. Type de composant (*Component Type*) : le type de composant représente les aspects configurables d'une implémentation.
- Propriété (*Property*) : ce sont les propriétés de configuration d'une implémentation (équivalent à « Propriétés de configuration »).
- Référence (*Reference*) : une référence représente un service requis d'un autre composant par l'implémentation (équivalent à « Dépendances de Service requis »).

[9] http://www.osoa.org/display/Main/Service+Component+Architecture+Home
[10] *Open Service Oriented Architecture* : http://www.osoa.org

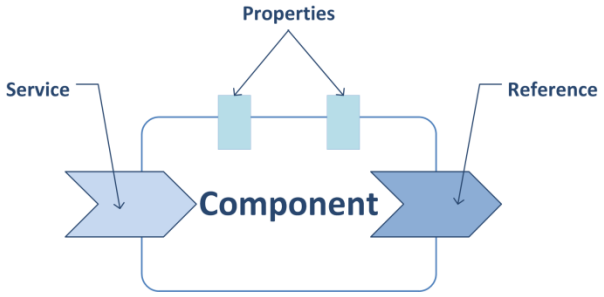


Figure 35 Composant orienté service dans SCA

Bien que l'avantage majeur de l'approche composant orienté service est de fournir les concepts et mécanismes pour la gestion de la disponibilité dynamique dans une application, la spécification de SCA ne le spécifie pas, laissant donc aux entreprises la possibilité de concevoir (ou non) des mécanismes ad'hoc.

De plus, SCA ne spécifie que quelques aspects non-fonctionnels, tels le déploiement et la configuration. Pour le reste, contrairement à iPOJO, SCA ne définit pas comment intégrer d'autres aspects non-fonctionnels tels la sécurité, le transactionnel… laissant donc aux entreprises la possibilité de concevoir (ou non) des mécanismes ad'hoc.

4.4.4 Comparaison

Table 2 Comparatif entre modèles à composant orienté service

	Service Binder	SCA	iPOJO
Hétérogénéité (langages…)	Non (Java)	Oui	Non(Java)
Dynamisme	Oui	Pas spécifié	Oui
Protocoles de communication	Java	SOAP, RMI,…	Java
Aspects non-fonctionnels	Non	Oui, prédéfini	Oui, ouvert (Handler)
Exécution distribuée	Non	Oui	Non
Spécification de service	Service	Service	Service
Implémentation de service	Composant à Service	Implémentation de composant	Composant à Service
Instance de Service	Instance de composant	Composant	Instance de composant

En résumé *Service Binder* comme *iPOJO* prend en charge la gestion de la disponibilité dynamique des services. De plus *iPOJO* supporte l'extensibilité du composant pour d'autres préoccupations. Contrairement à ces deux technologies SCA ne spécifie pas l'aspect dynamique laissant cet aspect au concepteur de la plate-forme, de même pour l'extensibilité SCA spécifie des préoccupations prédéfinis sans clairement interdire ou affirmer la possible extensibilité à d'autres préoccupations. En fait SCA et les deux autres technologies sont à des niveaux différents, *Service Binder* et *iPOJO* sont des technologies composant orienté service **à l'exécution**. Alors que SCA est une approche **pour la conception** d'architecture de composants orienté service. En d'autres termes ils ne sont pas concurrent, iPOJO et Service Binder pouvant être utilisés pour implanter une plate-forme SCA.

4.4.5 Synthèse

L'approche composant orienté service introduit la disponibilité dynamique dans un modèle à composant permettant ainsi aux applications la possibilité de s'adapter en fonction. Cette approche est

issue de la convergence des modèles à composant et de l'approche à service où les fonctionnalités d'un logiciel sont implantées par des composants et où la résolution de leurs dépendances ainsi que la communication suivent l'approche orientée service. Cette approche permet donc de fournir l'ensemble des propriétés fournies par [MN97] qui sont :

- La rapidité de la mise sur le marché (*Fast time-to-market*) ;
- La fiabilité ;
- La division du travail ;
- La variabilité ;
- La distribution ;
- La prise en charge de l'hétérogénéité ;
- L'adaptabilité.

5. SYSTEMES ADAPTATIFS

L'émergence de l'informatique ubiquitaire a changé le contexte d'exécution qui était établi jusqu'à présent car elle implique des équipements disséminés dans le monde réel ayant une disponibilité dynamique (cf. section 2 de ce chapitre). L'utilisation de ces équipements impacte l'architecture des applications qui subissent les variations de leur disponibilité. Ces modifications peuvent nécessiter l'intervention de manageur pour adapter l'architecture courante.

De nombreux projets cherchent à apporter une solution au problème de l'adaptation des architectures en cours d'exécution. Dans cette section, nous étudierons brièvement trois projets de recherche qui couvrent à eux trois les différentes approches et problématiques actuelles de l'adaptabilité. Ces projets seront caractérisés en fonction des critères suivants :

- **Environnement :** ce critère indique si l'environnement est centralisé ou réparti indépendamment de l'application.
- **Paradigme :** ce critère désigne le paradigme utilisé pour l'implémentation ou la composition de l'application. Par exemple : le paradigme composant.
- **Type d'application :** ce critère désigne le type de composition utilisé pour définir l'application. Par exemple : composition comportementale (orchestration).
- **Système ciblé :** ce critère indique quels types de logiciels et ou de contextes sont ciblés par le projet. Par exemple : l'auto-configuration du serveur JOnAS dans les grilles de calcul, ou la composition dynamique d'équipements ubiquitaires.
- **Répartition des éléments de l'application :** ce critère indique si les éléments du système sont centralisés, décentralisés (administrables) ou disponibles dynamiquement. Par exemple : composant local, composant administrable par JMX ou équipement UPnP.
- **Type d'adaptabilité :** ce critère définit le type d'adaptation supporté parmi les quatre catégories : auto-réparation, auto-configuration, auto-optimisation et auto-protection (cf. Figure 12).
- **Sondes et effecteurs :** ce critère indique quels aspects sont instrumentés : par exemple l'architecture, des sondes sur le ou les processeurs
- **Opérations d'administration :** ce critère indique quelles sont les primitives d'opération d'administration pour l'adaptation fournies par l'environnement.

Les projets retenus sont Rainbow, JADE et WComp.

Rainbow

Rainbow est un projet initié en 2004 (première publication) [CHGAl04] [CheT08]. Le but de ce projet est de pouvoir adapter l'architecture d'un système en cours d'exécution. Pour cela il fournit une plate-forme et un langage pour définir des systèmes auto-adaptables.

L'architecture de Rainbow définit un ensemble d'éléments pour la création d'applications autonomiques. Rainbow propose aux développeurs de baser l'ensemble du raisonnement d'adaptation sur un modèle de l'architecture courante du système. Pour cela chaque système devant être supervisé par Rainbow est instrumenté au moyen de sondes et d'effecteurs qui servent à modéliser le système en cours d'exécution. Cependant ces informations sont spécifiques à la plate-forme cible. Pour compenser, Rainbow propose une infrastructure de traduction vers le modèle à composant de Rainbow.

La particularité de ce projet est que les mécanismes de raisonnement et d'adaptation sont basés sur un modèle de l'architecture en cours d'exécution. Ce qui permet d'être générique et de réutiliser les mécanismes d'adaptation dans d'autres systèmes. Cependant sa principale limitation est le pendant de son avantage : le raisonnement et les choix d'adaptations sont uniquement basés sur l'architecture.

Table 3 Caractéristiques de Rainbow

Propriété	Rainbow
Environnement	Centralisé
Paradigme	Composant
Type d'application	Composition structurelle
Système ciblé	Générique (abstraction du système en exécution à l'aide d'un modèle)
Répartition des éléments de l'application	Dépend de l'environnement d'exécution sous-jacent.
Type d'adaptabilité	Surtout auto-réparation, mais l'auto-configuration est possible
Sondes et effecteurs	Seulement sur l'architecture
Opérations d'adaptation	Installation, instanciation, destruction et désinstallation de composant et modification des liaisons entre composants

JADE

JADE est un projet initié en 2005 (première publication) [BBHAl05] [DBBAl08]. Jade est un environnement pour l'administration autonome d'infrastructures logicielles patrimoniales ciblant en particulier la reconfiguration de serveurs JavaEE (JOnAS, JBoss…) sur des grilles / grappes de nœuds. Pour cela JADE propose d'encapsuler les logiciels patrimoniaux dans des composants respectant une interface d'administration homogène. L'un des principaux objectifs de JADE est d'augmenter la disponibilité (auto-réparation) des applications en allouant les ressources de façon optimale (auto-optimisation) tout en facilitant le passage à l'échelle pour l'administration d'applications de grande taille.

Dans JADE, l'application est vue comme une composition structurelle de composants administrables. La composition comme la configuration peuvent évoluer dans le temps. L'application est représentée dans un modèle à composant Fractal proposant des mécanismes d'introspection et de réflexion nécessaires à l'adaptation.

Le raisonnement et l'adaptation sont basés principalement sur l'architecture en cours d'exécution. Tout comme Rainbow, JADE permet le déploiement d'applications, cependant il permet en plus la découverte dynamique des nœuds et la gestion d'allocation des ressources.

Contrairement à Rainbow, JADE peut avoir plusieurs gestionnaires d'adaptation gérant des aspects distincts de l'application. Ces gestionnaires doivent être orthogonaux car JADE ne sait pas gérer les conflits entre gestionnaires.

Table 4 Caractéristiques de JADE

Propriété	JADE
Environnement	Réparti
Paradigme	Composant (Fractal)
Type d'application	Composition structurelle
Système ciblé	Logiciel patrimonial (testé en particulier sur les serveurs JavaEE) / pas de disponibilité dynamique des composants
Répartition des éléments de l'application	Réparti : composants distribués sur des serveurs JavaEE
Type d'adaptabilité	Auto-réparation et auto-optimisation
Sondes et effecteurs	Architecture, contexte logiciel et matériel
Opérations d'adaptation	Installation, instanciation, destruction et désinstallation de composants

	et modification des liaisons entre composants

WComp

Le projet de recherche WComp a été initié en 2003 (première publication) [FHLAl09] [TLRHR09]. Le domaine ciblé par ce projet est l'informatique ubiquitaire : les auteurs proposent une approche et un environnement qui permettent de définir des applications en terme de compositions structurelles à base d'équipements ubiquitaires, plus précisément à base d'équipements UPnP [UPnP-S] et DPWS [DPWS-S]. Dans ce domaine, une contrainte importante est que le processus d'adaptation se fait en fonction de l'environnement.

Le paradigme utilisé pour définir et administrer l'application est le paradigme *Service Lightweight Component Architecture* (SLCA) [HTLRR08]. Ce paradigme est une approche composant orienté service (cf. section 65 de ce chapitre) où le composant est un conteneur léger dans lequel se trouve le code d'adaptation ainsi que le client spécifique de l'équipement ubiquitaire qu'il représente. Deux points :

- Le composant dans une application WComp est un client de service. L'instanciation du composant se fait en fonction de la disponibilité dynamique du service qu'il représente, et non en fonction des besoins d'un administrateur.
- L'implantation de la logique d'adaptation est dans le conteneur du composant et est définie selon ce que les auteurs appellent des aspects d'assemblage (*aspects of Assembly* ou AA).

Les aspects d'assemblage sont des pièces d'information décrivant comment un assemblage de composants peut être structurellement modifié, tout en gardant la propriété « boite-noire » des composants. Ces modifications incluent l'ajout de composants et les liaisons entre eux.

Table 5 Caractéristiques de WComp

Propriété	WComp
Environnement	Centralisé
Paradigme	Intégration d'équipements ubiquitaires : UPnP et DPWS
Type d'application	Orchestration en SLCA (*Service Lightweight Component Architecture*)
Système ciblé	adaptation de l'orchestration en fonction de la disponibilité des services
Répartition des éléments de l'application	L'orchestration est centralisée mais les équipements utilisés sont distribués.
Type d'adaptabilité	Adaptation de l'orchestration en fonction de la disponibilité des services
Sondes et effecteurs	Code d'adaptation sur le conteneur réagissant à la disponibilité du service qu'il prend en charge donc pas de modélisation.
Opérations d'adaptation	Modification de l'orchestration

En résumé, l'application est construite par opportunisme, c'est-à-dire en fonction de la disponibilité des services UPnP et DPWS conformément à la définition de l'application faite en SLCA. L'adaptabilité de l'application est basée uniquement sur la disponibilité dynamique des services. De ce fait, les mécanismes d'adaptation rentrant en jeu ne nécessitent pas la vision globale de l'architecture en cours d'exécution, conséquemment l'environnement d'exécution ne modélise pas le système. En effet le but est centré sur la réactivité.

> *"We propose an approach for self-adaptive application in the field of ubiquitous computing based on models, in which adaptation is a reactive mechanism"[…] "Other works have explored the use of models at runtime for aspect-based adaptation "[…] "They need the dynamicity to validate new adaptation rules at runtime, before making them automatic scripts. But because our mechanism is build on less steps (especially model transformations) we seem more able to be reactive."*
>
> Extrait de [FHLAl09]

Synthèse

Les trois projets précédents n'ont ni les mêmes but ni les mêmes besoins. Cependant comme dans tous les projets visant à adapter l'environnement en cours d'exécution, la nécessité d'adaptation vient soit du besoin d'auto-réparation (Rainbow, JADE) soit du besoin de prendre en charge la disponibilité dynamique des éléments dans l'environnement (WComp).

Rainbow comme JADE essayent de modifier l'architecture en cours d'exécution **pour satisfaire les besoins de l'application**, alors que WComp modifie l'état de l'application en fonction de la **disponibilité dynamique** des éléments.

Pour pouvoir modifier l'architecture en cours d'exécution, Rainbow et JADE requièrent le modèle de l'application souhaitée ainsi que l'architecture en cours d'exécution. Ils abstraient l'architecture en cours d'exécution à l'aide d'un modèle mais pas pour les même raisons. Rainbow cherche à abstraire pour pouvoir réutiliser les mécanismes de raisonnement générique (indépendant de l'environnement d'exécution) ; alors que JADE abstrait l'architecture pour pouvoir masquer **l'hétérogénéité technologique** dans le domaine des serveurs JavaEE..

WComp cherche à réagir à la disponibilité des services UPnP et DPWS, ce qui ne requiert pas la vision globale de l'architecture en cours d'exécution.

En résumé, le but de Rainbow est de permettre l'établissement de mécanisme d'adaptation, pour cela il met en place sa boucle de rétroaction sur un modèle générique – c'est-à-dire non spécifique au domaine – de l'architecture en cours d'exécution. JADE cherche, quant à lui, à augmenter la disponibilité d'une application répartie sur des serveurs hétérogènes ; pour cela il abstrait l'hétérogénéité technologique des serveurs à l'aide d'un modèle sur lequel il base le raisonnement et les actions de l'adaptation. Le but de WComp est de mettre à jour l'architecture de l'application en réaction à la disponibilité dynamique des services UPnP et DPWS.

Cette thèse cherche à couvrir ces trois aspects pour les architectures orientées service. C'est-à-dire, fournir une plate-forme qui :

- Cache l'hétérogénéité des technologies orientées service ;
- Réagisse à la disponibilité dynamique des éléments qui la composent ;
- Soit générique et spécialisable pour pouvoir définir des mécanismes d'adaptation en fonction d'un domaine métier.

Chapitre 3 - Contribution

Les besoins actuels tendent vers l'interconnexion des systèmes d'information d'entreprises avec des équipements issus de l'informatique ubiquitaire. Les applications résultantes constituent un réel défi car elles ont à la fois une partie de leur architecture qui est statique alors que l'autre est dynamique, une partie des éléments de l'architecture qui sont administrables, et l'autre non … En d'autres termes, l'architecture de ces applications est constituée d'éléments ayant des comportements très divers et dont la disponibilité peut être non déterministe.

L'objectif à terme est de contrôler l'évolution de telles architectures. Pour cela, il est nécessaire d'avoir une approche globale : c'est-à-dire de définir une approche couvrant l'ensemble des aspects nécessaires à l'exécution de telles applications. Dans notre cas, cette approche couvre les phases de conception, de déploiement et d'exécution du cycle de vie de l'application (cf. Chapitre 2 - section 1). Nous souhaitons transférer une partie de la connaissance des éléments et de l'application à la phase d'exécution, pour pouvoir diriger l'adaptation (cf. Chapitre 2 - section 2) de l'architecture. L'adaptation passe par la comparaison de l'état de l'architecture en cours (propriétés intrinsèques et contextuelles des éléments) avec la définition (*Connaissance*) de l'application. Pour cela, d'une part nous abstrayons l'état d'exécution sous la forme d'un modèle (cf. Chapitre 2 - section 3) et de l'autre nous définissons un modèle d'architecture de l'application par intention. Dans notre cas ces deux modèles partagent un sous-ensemble de leur méta-modèle définissant le concept de composant orienté service (cf. Chapitre 2 - section 4). Le but recherché dans notre approche est de permettre à la fois l'adaptation de l'architecture en cours d'exécution en fonction des besoins de l'application, comme les projets JADE et Rainbow (cf. Chapitre 2 - section 5), et l'adaptation de l'application en réaction à la disponibilité dynamique des éléments qui la composent, comme WComp (cf. Chapitre 2 - section 5).

Cette thèse propose un environnement d'exécution dirigé par les modèles pour la description et l'adaptation d'architecture à base de services hétérogènes.

Dans ce chapitre, nous aborderons dans un premier temps l'approche globale dans laquelle s'inscrit cette thèse. Nous passerons ensuite à la conception du méta-modèle en définissant les différents concepts cœurs ; puis nous définirons pour chaque concept les différentes opérations et propriétés qu'ils ont ou peuvent avoir à l'exécution. Nous traiterons ensuite trois aspects de notre environnement d'exécution que sont la distribution, l'intégration de services hétérogènes et l'extensibilité à d'autres préoccupations.

1. UNE APPROCHE GLOBALE

Pour satisfaire les besoins actuels, il faudrait avoir des environnements d'exécution qui s'adaptent aux changements du contexte et aux besoins non-fonctionnels. Cependant, le contexte d'exécution est un concept diffus, il faut le limiter aux aspects pertinents. Or le contexte d'exécution varie au cours du temps. Par conséquent, définir une application statiquement en prenant en compte l'ensemble « pertinent » des contextes pour son exécution est extrêmement difficile. En effet l'ensemble des aspects définissant le contexte d'exécution peut être infini. De plus, dans le cas d'une définition statique, l'ajout d'un aspect au contexte d'exécution nécessite un nouvel incrément du cycle de développement empêchant ainsi l'adaptabilité dynamique à l'exécution.

Cette problématique requiert une approche globale depuis le développement jusqu'à l'exécution pour garantir la continuité de la connaissance, ainsi que d'un environnement homogène.

1.1 L'APPROCHE SAM

L'**objectif** de l'approche SAM est de définir et d'exécuter des applications pouvant être adaptées à l'exécution. À partir d'une définition de l'application, l'environnement d'exécution doit-être capable d'interpréter et « d'exécuter » cette application.

Cet objectif est simple à exprimer, mais du fait de sa généralité il est extrêmement complexe à réaliser. En effet, l'approche que nous proposons se veut applicable à toutes les applications orientées service. Nous devons être ouverts aux différentes méthodologies de développement, aux différentes propriétés non-fonctionnelles, aux différents modèles métier... Pour cela, nous fournissons une approche générique que l'on peut spécialiser pour chaque domaine métier, ce qui impose de :

- Définir le cycle de vie d'une application en un ensemble de phases ;
- Permettre la définition d'applications « statiques » et « dynamiques » ;
- Permettre l'extensibilité de la définition de l'application avec des contraintes / propriétés non-fonctionnelles ;
- Formaliser la définition de l'application et de ses extensions suivant un langage homogène et abstrait ;
- Exécuter de telles applications ;
- Garantir la cohérence de l'exécution vis-à-vis de la définition de l'application (aspects non-fonctionnels inclus).

Nous allons dans un premier temps nous intéresser aux cycles de vie dans l'approche SAM, puis à l'architecture. Une fois le contexte posé, nous définirons l'objectif de cette thèse.

1.2 LES PHASES DU CYCLE DE VIE

L'approche SAM ne propose pas de méthodologie de développement et laisse ce choix aux responsables des domaines métier. Cependant l'approche SAM préconise la transmission à l'exécution de la C*onnaissance* et des C*hoix* qui ont été faits lors du développement afin qu'ils soient pris en charge. La *Connaissance* est un concept vaste, regroupant toutes les informations pour tous les aspects dont on dispose pour un élément. Dans notre cas, nous limiterons le concept de *Connaissance* aux informations de définition d'une application, c'est-à-dire aux informations sur l'architecture la configuration, les aspects non-fonctionnels… De même, nous limiterons le concept de *Choix* à l'aspect architectural de l'application. Les informations comme : pourquoi avoir utilisé une technologie plutôt qu'une autre pour le développement d'un composant, ne sont pas pertinentes de notre point de vue car il n'y a pas de sens de les remettre en cause à l'exécution.

Par conséquent, il faut éliminer le cloisonnement entre l'environnement de développement et l'environnement d'exécution, pour que l'ensemble des informations (la connaissance) soient « propagées » d'activité en activité.

Bien que l'approche SAM ne propose pas de méthodologie, elle définit tout de même trois grandes phases : le cycle de vie de développement (cf. Chapitre 2 - section 1.1), le cycle de vie du déploiement (section 1.2) et le cycle de vie de l'exécution (section 1.3).

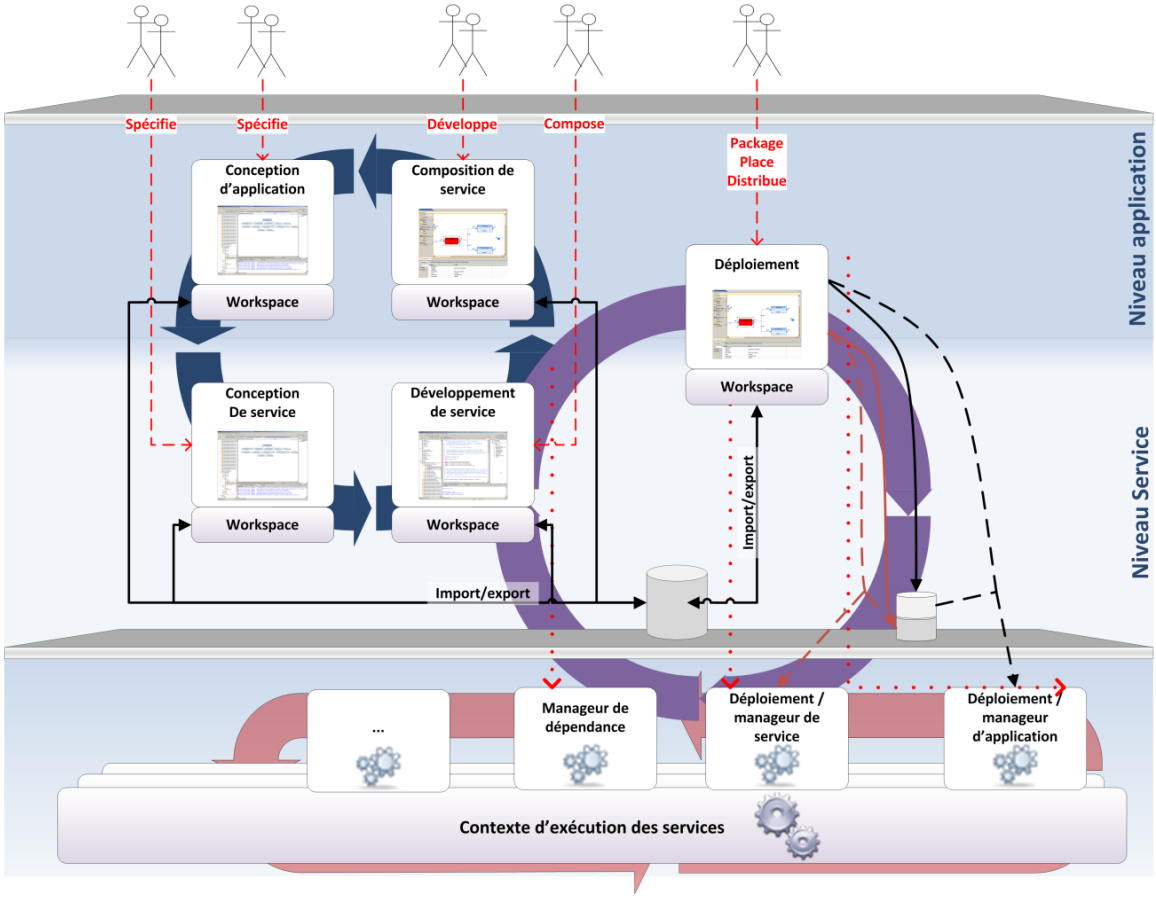


Figure 36 Cycles de vie dans l'approche SAM

Nous répartissons les activités dans ces trois cycles de la manière suivante :

Cycle de vie de développement

L'ensemble des phases en amont du développement (en termes de code) d'un composant logiciel et de la définition d'une application se trouvent dans le cycle de vie de développement. Dans notre approche, la frontière entre les cycles de vie de développement et du déploiement est le dépôt de « code ».

Cycle de vie du déploiement

L'administrateur en charge du déploiement possède, d'une part la *connaissance* de l'application grâce au dépôt de « code », et de l'autre la connaissance de l'infrastructure, qui sont nécessaires aux activités de packaging, de placement, de configuration, de provisionnement des dépôts… L'activité frontière entre le cycle de vie du déploiement et celui de l'exécution est l'envoi du modèle d'application aux environnements d'exécution.

Cycle de vie de l'exécution

Dans notre cas, exécuter une application consiste à maintenir l'application dans un état *cohérent, fonctionnel* et *optimal* en fonction de l'évolution du contexte durant toute la phase d'exécution. Pour ce faire, le mécanisme assurant l'exécution de l'application compare l'état d'exécution (*Connaissance* de l'exécution) avec la définition de l'application (*Connaissance* de l'application) et agit (*Choix*) en fonction des solutions possibles. La phase d'exécution a donc bien un cycle de vie constitué d'un ensemble d'activités autres que la supervision par un administrateur humain.

Connaissance et Choix

Dans notre cas, la *Connaissance* de l'application se traduit par un modèle d'application étendu par les concepts et les métadonnées des contextes métier ciblés. La *Connaissance* de l'exécution se traduit par un modèle de l'état d'exécution étendu par les contextes opérationnels ciblés. Par conséquent, il est possible de refaire des choix à l'exécution et donc de refaire un certain nombre d'activités habituellement faites au développement. Dans notre cas, les choix pouvant être refaits sont ceux définissant la configuration ou l'architecture de l'application.

L'interaction entre les cycles se retrouve dans l'architecture de l'environnement SAM.

1.3 ARCHITECTURE DE L'ENVIRONNEMENT SAM

L'approche SAM se base sur de nombreux travaux au sein de l'équipe ADELE et en est la source d'autres. Ces travaux se situent à différentes phases du cycle de vie de l'application.

L'approche SAM fournit un environnement homogène couvrant le cycle de vie du développement à l'exécution d'applications dynamiques ouvertes au contexte. Cet environnement est lui-même scindé en deux environnements complémentaires et partiellement concurrents :

- l'environnement de développement - constitué d'un sous-ensemble d'environnements de développement spécialisés pour des activités/contextes - qui définit l'application et les services (*Connaissance* de l'application) et qui prédéfinit et/ou contraint les choix dans la variabilité de l'application (*Choix*) ;
- l'environnement d'exécution qui « finalise » les choix et maintient l'état de l'application en fonction de l'état d'exécution (*Connaissance* de l'exécution) conformément à la définition de l'application (*Connaissance* de l'application).

Dans l'approche orientée service, nous pouvons distinguer deux éléments : l'application et les services.

Le développement d'un service est indépendant de son utilisation / sa composition dans une application. Le service est développé pour fournir une fonctionnalité. Cette séparation reste vrai à l'exécution (particulièrement dans l'approche orientée Web service (cf. Chapitre 2 - section4.3)), le service s'exécute sans avoir connaissance des applications auxquelles il appartient.

De même il faut noter qu'une application est un composite [DieT10] d'éléments atomiques et/ou composites, alors qu'un service est considéré par SAM comme un élément atomique. Pour SAM, un service, en tant qu'élément atomique, a un comportement dynamique limité : il est disponible ou ne l'est pas ; conséquemment il ne nécessite pas d'être adapté dynamiquement. A contrario, un composite peut être constitué d'éléments ayant une disponibilité dynamique. Le comportement dynamique des éléments impose de vérifier continuellement la validité du composite.

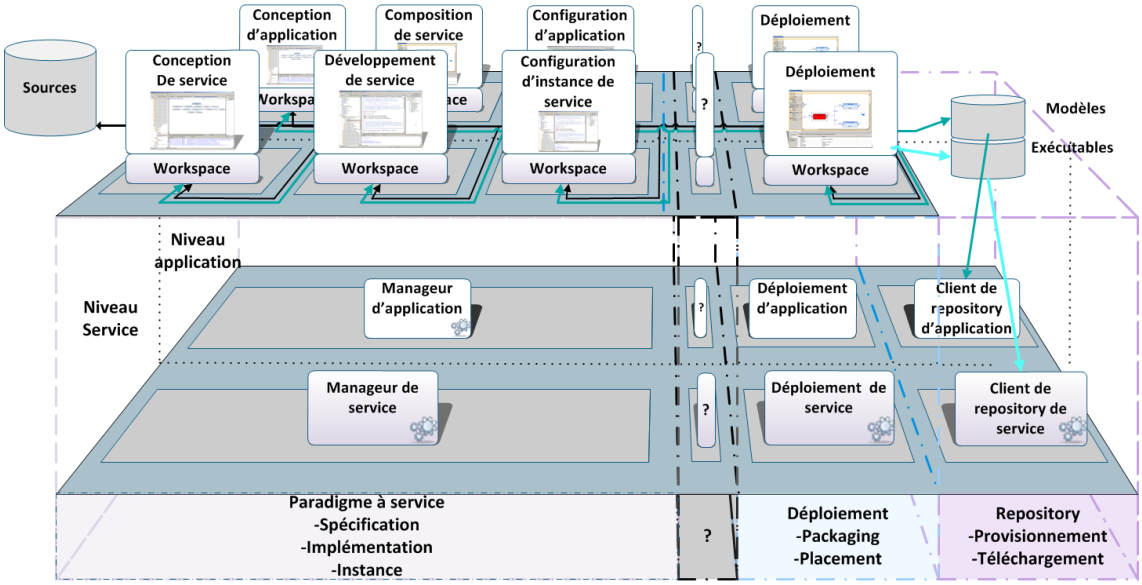


Figure 37 Architecture de l'environnement SAM

La Figure 37 schématise l'architecture de l'environnement SAM. En haut nous avons les environnements de développement nécessaires au développement d'applications orientées service. Dans notre cas, chaque environnement correspond à un CADSE [EVLL08] [CADSE]. Ces CADSEs sont synchronisés via une base de versionnement [LevT10] [ELV09]. Un administrateur en charge du déploiement de service et / ou d'application utilise cette base de versionnement pour packager, configurer,… avant de provisionner le dépôt correspondant. L'administrateur envoie l'ordre aux environnements d'exécution de charger leur modèle d'application.

Il y a une similarité entre l'environnement d'exécution et celui de développement. En effet, si l'environnement de développement produit des services alors l'environnement d'exécution sera un SOA ; si l'application est définie comme une orchestration, alors l'environnement d'exécution possèdera un moteur d'orchestration. Le fait de savoir que l'on a une orchestration parce que l'environnement d'exécution possède un moteur d'orchestration, ou que l'on a un moteur d'orchestration parce qu'on a une application définie par orchestration importe peu : ce qui est intéressant c'est que **l'environnement d'exécution correspond à la projection de l'environnement de développement**. Un service est développé pour une technologie donnée que possède l'environnement d'exécution. Une application est définie selon une méthode : orchestration, chorégraphie ou composition structurelle, compréhensible à l'exécution via un manageur d'application correspondant à la méthode…

L'environnement d'exécution est scindé entre un niveau « exécutif » qui fournit l'état d'exécution *correct* (cf. Chapitre 2 - section 3.3) et les opérations primitives des services, et un niveau « décisionnel » qui vérifie que l'état d'exécution (fourni par le niveau « exécutif ») est conforme à la définition de l'application. Le niveau décisionnel exécute des opérations pour rétablir la *validité* (cf. Chapitre 2 - section 3.3) si nécessaire.

Nous voyons que la problématique est vaste et que la solution proposée est complexe. En d'autres termes, cette thèse ne peut pas et n'a pas pour but de résoudre toutes les problématiques de l'approche ci-dessus.

1.4 OBJECTIF DE LA THESE

Cette thèse a pour but de définir un environnement d'exécution – nommé SAM-RT – qui modélise l'état d'exécution de la plate-forme. Ce modèle doit :

- Cacher l'hétérogénéité des technologies orientées service ;
- Réagir à la disponibilité dynamique des éléments qui la compose ;
- Etre générique et spécialisable pour pouvoir définir des mécanismes d'adaptation en fonction d'un domaine métier.

Nous proposons de résoudre ces trois aspects en :

- Définissant un méta-modèle de service offrant les concepts nécessaires pour l'adaptation dynamique des applications ;
- Définissant un mécanisme d'intégration des technologies SOA dans un modèle homogène conforme au méta-modèle défini précédemment ;
- Définissant un mécanisme d'intégration de nouvelles préoccupations ;
- Définissant le moyen d'interaction entre le niveau « exécutif » (service) et le niveau « décisionnel » (application).

Conséquemment dans la suite de ce chapitre, nous allons dans un premier temps (1) définir et justifier le méta-modèle de l'approche SAM. Puis (2) comment il est utilisé à l'exécution. Nous allons ensuite expliquer comme (3) gérer la répartition de l'application. Enfin nous aborderons (4) l'intégration des services issus de différentes technologies SOA via le méta-modèle défini précédemment et (5) comment étendre le contexte d'exécution. Et nous finirons (6) par l'utilisation du modèle réflexif obtenu par les manageurs d'application et d'adaptation.

2. META-MODELE DE SERVICE DANS SAM

Cette section aborde le méta-modèle de Service défini dans l'approche SAM. Le méta-modèle SAM se base sur un mécanisme nommé **groupe d'équivalence**. Nous allons définir ce mécanisme avant d'aborder le méta-modèle en lui-même.

2.1 GROUPE D'EQUIVALENCE

Le concept de groupe d'équivalence est transversal aux différents travaux utilisés dans l'approche SAM. Nous résumerons les caractéristiques de ce concept pour mieux appréhender le méta-modèle SAM. [LevT10] [DieT10] et [EDSM10] expliquent plus en détail le concept de groupe d'équivalence.

Un **groupe d'équivalence** (ou plus simplement **groupe**) est défini comme un ensemble d'éléments qui sont indiscernables pour un certain point de vue. Un groupe est constitué d'un objet **représentant** et d'un ensemble d'objets **membres** du groupe. Le **point de vue** du groupe est défini par un ensemble de propriétés (et de relations) que possèdent tous les membres du groupe correspondant (cf. Figure 38). Ces propriétés sont celles du représentant.

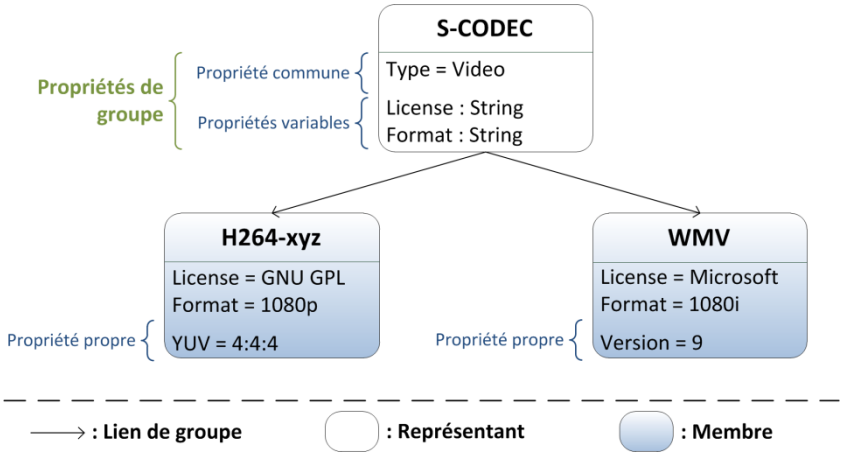


Figure 38 Groupe d'équivalence

Une contribution majeure de la notion de groupe est que le représentant ne détient pas que les propriétés communes du groupe, mais il définit également les propriétés par lesquelles les membres peuvent être distingués. Le représentant définit donc :

- Les **propriétés communes** : celles-ci - clé et valeur - sont propagées du représentant à chacun des membres. Tous les éléments du groupe partagent les propriétés communes définies par le représentant (cf. « Type = Video » de la Figure 38).
- Les **propriétés variables** : c'est la **définition** des propriétés que doivent nécessairement avoir chaque membre. Si le représentant définit une propriété variable alors tous les membres du groupe doivent donner une valeur pour cette propriété ; cependant sa valeur est fixée par le membre et non par le représentant.
- Les membres d'un groupe peuvent aussi définir des propriétés qui leur sont propres. Celles-ci s'appellent **propriétés propres**.

La Table 6 résume qui du représentant ou du membre définit la clé et la valeur des propriétés du groupe.

Table 6 Résumé des définitions de clés et valeurs des propriétés de groupe

	Clé	Valeur
Propriété commune	Représentant	
Propriété variable	Représentant	Membre
Propriété propre	Membre	

On peut considérer le représentant comme le type des membres de son groupe.

2.2 COMPOSANT A SERVICE VS. COMPOSANT ORIENTE SERVICE

Dans les composants à Service (cf. Chapitre 2 - section 4.3), la séparation entre services et composants fait qu'à l'exécution l'application a une vue « utilisateur » des services et ignore les détails d'implémentations, tandis que les composants ont une vue « fournisseur » qui ne tiennent pas compte du niveau application. Cette séparation fait qu'il n'est pas possible de définir une application par composition structurelle, seule la composition comportementale est possible. Or, la composition comportementale permet seulement de reconfigurer l'application en substituant un service par un autre, tout en ignorant les éléments et leurs dépendances. En effet la composition comportementale ne permet pas de déployer un nouveau service. Les services ont été conçus pour isoler l'application des préoccupations d'implémentations, ce qui rend difficile les changements et l'administration des implémentations et autres aspects liés à l'exécution, que requièrent les plates-formes orientées service dynamiques et adaptables (cf. la section « *Research Challenges* » page 41 de [PTDL07], [BBF09] et [FHSAI06]).

Pour «permettre le dynamisme, une application doit […] permettre la modification de son architecture » [EscT08]. En d'autres termes : pour garantir la conformité de l'architecture d'une application utilisant des éléments logiciels disponibles dynamiquement, celle-ci doit pouvoir être modifiée. Cependant pour la modifier à l'exécution, il est nécessaire de connaitre les dépendances des instances utilisées dans l'application, ainsi que les opérations d'administration permettant d'effectuer les modifications. L'approche composant orienté service (cf. Chapitre 2 - section 4.4) résout ces différents problèmes. En effet dans cette approche, les concepts de service et d'implémentation (composant) ne sont pas dissociés ; les composant fournissent des services et dépendent de services.

La Table 7 résume partiellement quelques points de vue liés à l'adaptabilité des applications. Il est intéressant de remarquer que les composants orientés service peuvent être utilisés par les orchestrateurs comme des services. En effet les composants orientés service sont des composants à service ayant des dépendances en termes de services. Ce qui les différencie c'est avant tout la composition. Un composant à service a un seul et unique but : fournir le service spécifié ; il est masqué du niveau application. Alors qu'un composant orienté service (dans le cas d'une composition structurelle) a une existence dans l'application. Par conséquent il peut être manipulé (modulo les opérations d'administration) ce qui permet l'adaptabilité à l'exécution.

Table 7 Comparaison partielle entre composant à service et composant orienté service

	Service et Composant à Service	Composant Orienté Service
Niveaux « Application » et « Implémentation »	Dissociés	Unifiés
Actions à l'exécution sur l'architecture de l'application	Substitution d'un service par un autre	Potentiellement toutes les opérations d'administration : installation, instanciation, substitution, destruction…
Composition	Comportementale	Comportementale et Structurelle

2.3 META-MODELE « CŒUR » DE L'APPROCHE SAM

L'approche SAM spécifie un environnement cohérent et homogène permettant la spécification, l'implémentation et l'exécution d'applications orientées service. Le « cœur » de l'approche SAM est un méta-modèle permettant :

- la définition d'applications statiques ou dynamiques ;
- la définition d'applications par extension ou par intention.

La sémantique du méta-modèle est identique de la conception à l'exécution. Ce méta-modèle se nomme *SAM CORE*. Nous distinguons dans *SAM CORE* trois concepts centraux que sont : la spécification d'un composant orienté service, l'implémentation d'un composant orienté service et l'instance d'un composant orienté service. Par abus de langage nous dirons par la suite « service » pour « composant orienté service ».

Pour illustrer notre propos, nous reprendrons l'exemple défini dans l'introduction du Chapitre 3 - : « *Cette entreprise souhaite aussi louer une passerelle mulimédia pour la maison. le but de cette passerelle est de découvrir les équipements domotiques Hifi, pour fournir les médias en exploitant les équipements présents. Par exemple, un client possède un téléviseur UPnP de résolution 1920*1080 pixel et un ampli DPWS en DTS 5.1. Ces équipement sont découverts par la passerelle multimédia de l'entreprise XYZ. Le client peut alors télécharger les médias sur sa passerelle multimédia dans son format optimal pour son équipement. Ce média peut être alors lancé, la passerelle se chargera de configurer la télévision et l'ampli* ».

2.3.1 Spécification

Dans notre exemple, à la phase de spécification, nous pouvons identifer trois concepts centraux : la passerelle multimedia, le téléviseur et l'ampli. Dans cet exemple, la passerelle multimédia est un client des services fournis par le téléviseur et l'ampli. Celle-ci dépend donc de ces deux services. De plus, le service multimédia de la passerelle peut d'un côté être administré par l'entreprise et de l'autre être utilisé par le client pour déployer un film, le lire…

Par conséquent nous définissons quatre spécifications de service :

- Le service ampli qui fournit une interface pour le son ;
- Le service téléviseur qui fournit une interface pour la vidéo et une pour le son ;
- L'interface utilisateur (cela peut être une télécommande, une page web, basée sur la reconnaissance des mouvements…) qui requiert le service de passerelle multimédia ;
- Et finalement la passerelle multimédia qui requiert un service de son et un autre de vidéo.

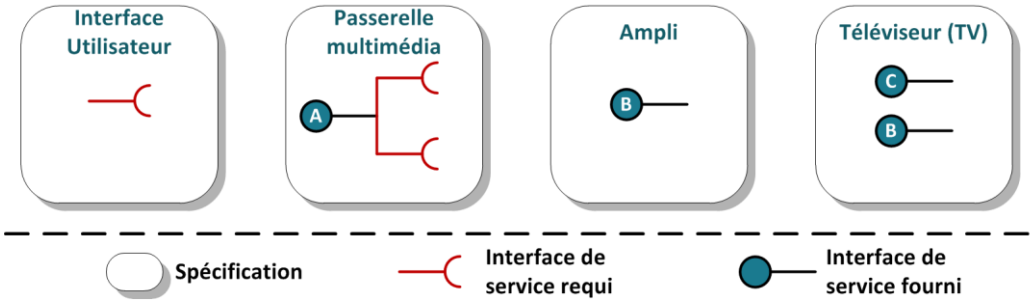


Figure 39 Spécification SAM des services de l'exemple

Dans notre approche un service n'est pas juste une interface d'invocation ou un « thème » (*topic*) d'envoi de données. Lors de la phase de spécification, un service est la spécification abstraite d'une fonctionnalité logicielle. Un composant orienté service fournit **zéro ou plusieurs interfaces** d'invocation. Contrairement aux autres approches orientées service, on définit les dépendances d'interface de service dès la spécification. Par conséquent, une spécification de composant orienté service peut avoir **zéro ou plusieurs dépendances** vers d'autres spécifications de service. Et finalement une spécification de service peut avoir **zéro ou plusieurs propriétés**. Notons que le méta-modèle de *SAM CORE* se veut indépendant de la façon de composer l'application. Il n'y a donc ni contraintes ni propriétés contextuelles ; ces concepts n'ont de sens que dans le contexte de l'application (cf. [DieT10]). Dans *SAM CORE*, les *propriétés* d'une spécification sont les propriétés intrinsèques du service spécifié.

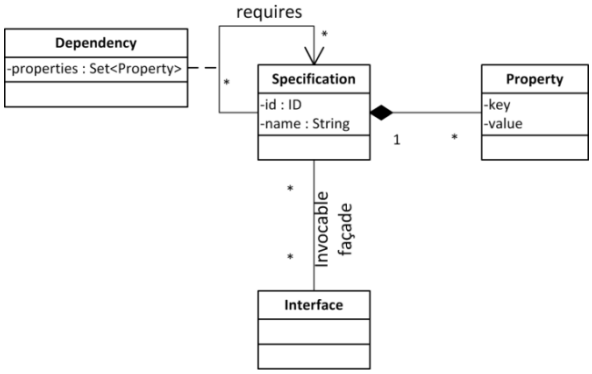


Figure 40 Concept de spécification

La Figure 40 modélise le concept de spécification.

2.3.2 Implémentation

Une spécification de service peut être implémentée de plusieurs façons. Une spécification peut avoir **zéro ou plusieurs implémentations**.

La relation qui unit une spécification et ses implémentations est la même relation qui unie un représentant et les membres de son groupe d'équivalence. La spécification est le type de l'implémentation : toutes les caractéristiques de la spécification sont propagées à ses implémentations. De la même manière qu'une classe et ses instances, les implémentations sont des instances **configurées** d'une spécification. Une implémentation **implémente une unique spécification** (pas d'« héritage multiple »). Une implémentation **doit implémenter** les interfaces de la spécification, et **doit au moins référencer** les dépendances de celle-ci, mais peut aussi en ajouter. De manière similaire, une implémentation doit **au moins** avoir les propriétés définies par la spécification, mais peut aussi **ajouter zéro ou plusieurs propriétés propres**.

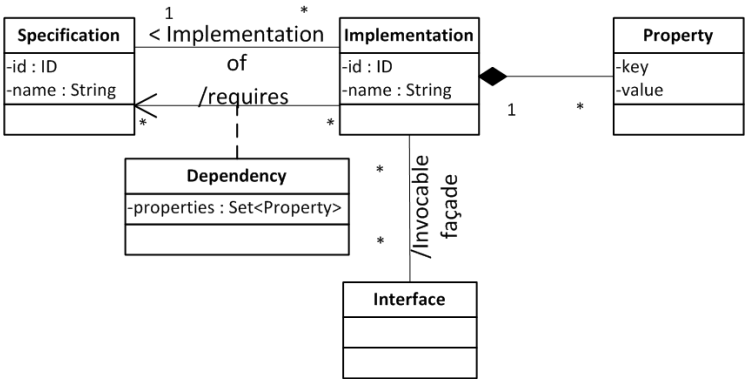


Figure 41 Concept d'Implémentation

La Figure 42 montre différentes implémentations possibles pour la spécification du service d'Interface Utilisateur (IU). Comme dit dans la section précédente, le service d'interface d'interaction avec l'utilisateur peut se faire de différentes façons. Dans l'exemple ci-dessous on a :

- Une implémentation pour les télécommandes de l'entreprise SAMSUNG© ;
- Une implémentation qui dépend d'un serveur Web pour exporter une page web d'interaction ;
- Une implémentation qui se base sur la reconnaissance de mouvement à l'aide d'une Webcam USB Logitech© ou Microsoft© (les pilotes sont embarqués).

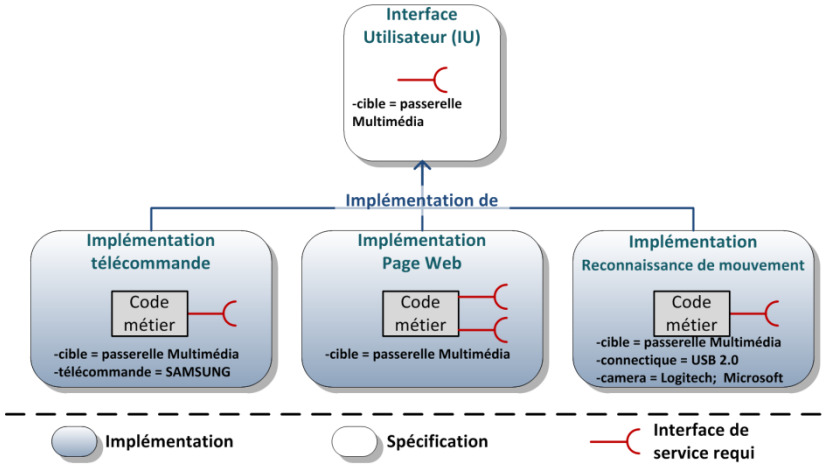


Figure 42 Exemple d'implémentations possibles pour la spécification "UI"

Notons que selon la logique de groupe, tous les membres doivent au moins avoir les propriétés communes, mais peuvent en ajouter d'autres. Donc, en théorie, une implémentation devrait pouvoir ajouter des interfaces supplémentaires. Cependant les dépendances sont en termes de spécifications ; et donc les interfaces ajoutées ne seraient jamais utilisées.

2.3.3 Instance

Une implémentation peut **avoir zéro ou plusieurs instances configurées** de service.

De même qu'entre une spécification et ses implémentations, la relation qui unit une implémentation et ses instances est une relation de groupe. Cependant, les groupes d'équivalence, bien que définissant un typage, ne définissent pas la nature de la relation « d'instanciation » entre un représentant et ses membres. De même qu'une implémentation est l'instanciation d'une spécification, une instance est l'instanciation d'une implémentation. Dans la plupart des cas une spécification est « instanciée » (développée) manuellement faisant appel à la *créativité* du développeur qui peut donc lui

ajouter des propriétés propres comme des dépendances … Alors que dans le cas d'une instance à l'exécution, celle-ci – exceptées les propriétés de configuration – a exactement les caractéristiques intrinsèques de son implémentation. Notons qu'une implémentation peut également être générée (instanciée) à partir d'une spécification configurée.

Une instance a, par propagation directe, les propriétés, les interfaces et les dépendances de son implémentation et donc par transitivité a également celles définies par sa spécification (propagées à l'implémentation). Une instance **a exactement les interfaces dérivées de sa spécification**, et les **dépendances dérivées de son implémentation**. On peut cependant lui ajouter des propriétés en plus de celles propagées.

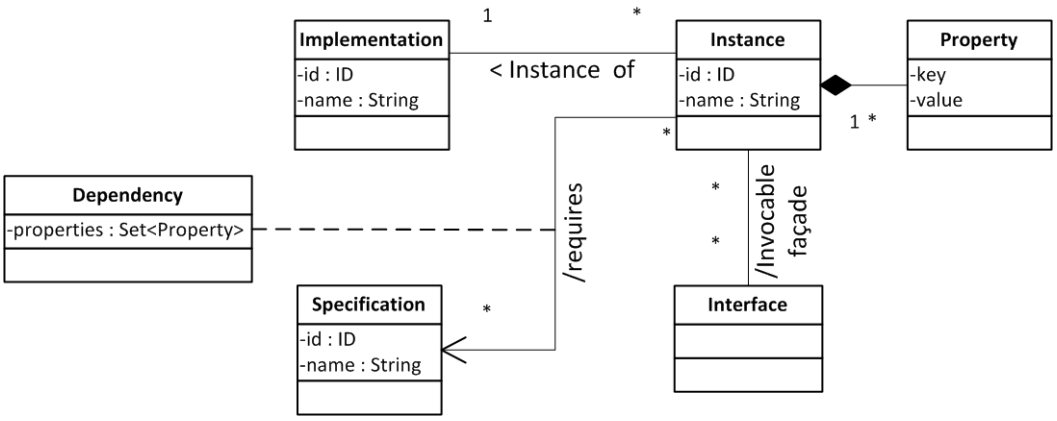


Figure 43 Concept d'Instance

2.3.4 Identification

Dans l'approche orientée service, le choix d'une instance de service se fait, en théorie, non pas par identification, mais par sélection en fonction des fonctionnalités et des propriétés telles que la qualité de service. Le service est donc utilisé de manière opportuniste et celui-ci n'est pas prédéfini durant la phase de spécification ou au développement. Par conséquent la configuration – c'est-à-dire les instances de service – de l'application est établie de manière opportuniste à l'exécution. Le nombre de configurations possibles est égal au nombre de combinaisons possibles avec les instances de services contenues dans le registre. Par conséquent garantir l'exécution de l'application obtenue équivaut à garantir l'ensemble des configurations, ce qui est extrêmement ardu. Par conséquent, il existe deux manières de faire :

- soit nous avons la maîtrise des registres de services et on fait en sorte qu'ils ne contiennent que les éléments que l'on désire ;
- soit il est possible d'identifier le service que l'on veut utiliser.

Le fait d'identifier un élément signifie que l'on est capable de le reconnaitre de façon certaine parmi un ensemble à l'aide une clé d'identification : soit par un identificateur unique, soit par un tuple de propriétés uniques. En pratique les applications ne choisissent pas arbitrairement une instance de service à l'exécution mais elles la choisissent lors de la définition de l'application. Ainsi, les entreprises utilisent rarement les annuaires UDDI[11] pour la sélection de Web-Service, préférant cibler – dans la configuration, voir dans le code – explicitement le Web-Service via son adresse.

[11] *Universal Description, Discovery and Integration*. http://uddi.xml.org/

De fait dans le méta-modèle *SAM CORE*, les services, que ce soient les spécifications, les implémentations ou les instances, sont identifiables ; prenons l'exemple de la Figure 44 :

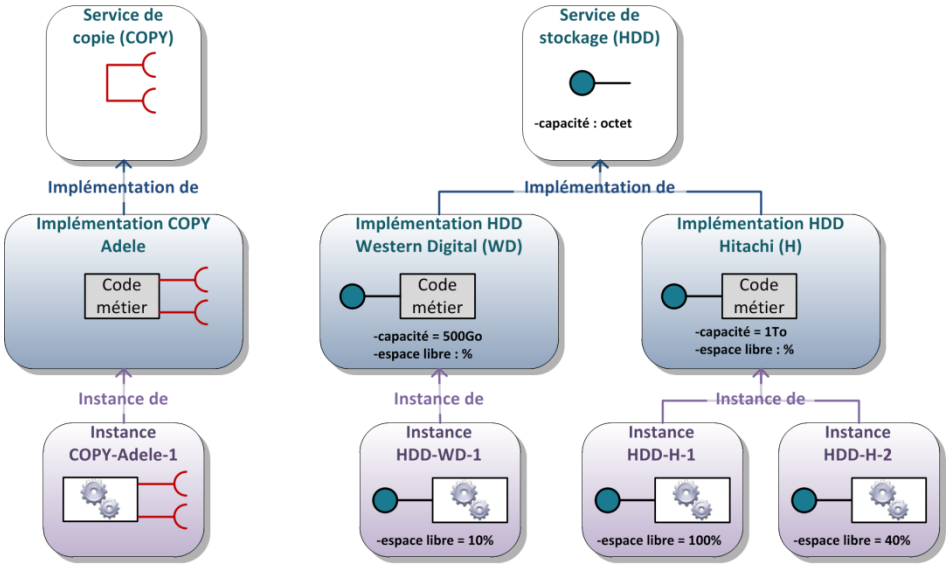


Figure 44 Exemple de problème d'identification

Dans l'exemple ci-dessus un service de copie permet de copier le contenu d'un service de stockage vers un autre. En d'autres termes, il permet de faire une copie de sauvegarde. La spécification du service de copie a donc deux dépendances vers des services de stockage : une dépendance du service de stockage que l'on veut sauvegarder, et une dépendance du service de stockage sur lequel on veut sauvegarder. Cet exemple expose deux problèmes d'identifications, le premier est qu'on ne veut pas sauvegarder n'importe quel service de stockage, ni n'importe où. Dans le cas de l'exemple, on veut copier le contenu de l'instance XDD-WD-1 vers HDD-H-1. On doit donc identifié les instances, cela peut aussi être vrai pour les implémentations et les spécifications. L'introspection du service de copie ne permet pas de discerner quelle dépendance requiert le service à sauvegarder et quelle dépendance requiert le service où les données seront sauvegardées. Par conséquent l'architecture obtenue par introspection est ambigüe.

Dans le méta-modèle *SAM CORE*, tous les éléments sont identifiables. La Table 8 résume les identificateurs des éléments (pour rappel l'environnement SAM est basé sur le langage Java) :

Table 8 Identificateur des concepts du méta-modèle de SAM CORE

Concept	Identificateur
Spécification	ID
Implémentation	ID
Instance	ID
Interface	Interface Java (donc package, classe et *classloader*)
Propriété (de service, de configuration, de dépendance)	Clé de la propriété (doit être unique)
Dépendance	ID

2.4 SYNTHESE : LE META-MODELE SAM CORE

Dans l'approche orientée service (cf. Chapitre 2 - section 4.2), et dans les modèles à composant (cf. Chapitre 2 - section 4.1), nous avons vu que les concepts de spécification, d'implémentation et d'instance sont ambigus. En fonction de la phase et de l'acteur, le concept de service sera implicitement rattaché à l'un de ces trois concepts.

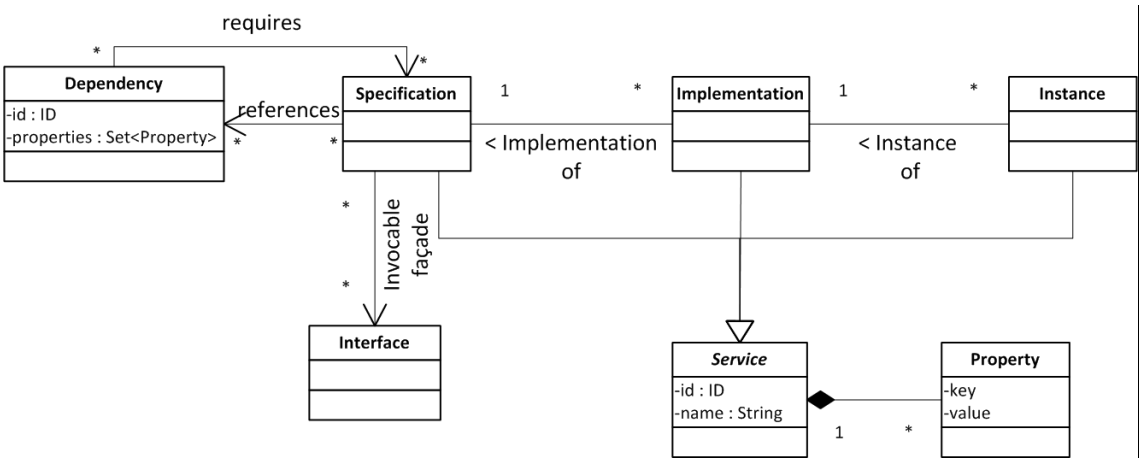


Figure 45 Méta-modèle de SAM CORE

Dans le cas de *SAM CORE*, nous parlons de spécification, d'implémentation et d'instance de service quelle que soit la phase du cycle de vie de l'application. Ces concepts ont une sémantique constante dans le cycle de vie de l'application. Cependant en fonction de la phase, la relation « imageDe » entre cet élément du modèle et le système étudié (cf. Chapitre 2 - section 3.3) peut ne pas (encore) exister. Par exemple, lors de la définition d'application, une « instance » est une déclaration d'instance alors qu'à l'exécution, cette même « instance » sera réellement l'image d'une instance réelle. Le méta-modèle de *SAM CORE* défini par la Figure 45 est la factorisation des différents méta-modèles définis précédemment prenant en compte la notion d'identification.

Nous appelons ce méta-modèle *SAM CORE*, car il est l'élément central de notre approche. Ce méta-modèle permet de définir – en tant que modèle prescriptif – le but fonctionnel d'une application. C'est-à-dire l'utilité du logiciel qui a pour but de résoudre un problème donné, alors que les aspects comme le déploiement, la sécurité, les transactions… ne définissent que des propriétés techniques : c'est-à-dire des propriétés que doit avoir une application d'un point de vue technique.

3. ENVIRONNEMENT D'EXECUTION

L'objectif de cette thèse est de définir et réaliser un environnement d'exécution dirigée par les modèles qui permet la description et l'adaptation d'architectures à service hétérogènes. Dont l'un des buts est de masquer les problèmes liés aux technologies utilisées pour ne se concentrer que sur les problématiques de l'application. De ce fait l'environnement d'exécution SAM (*SAM-RT*) masque les différentes technologies à services en abstrayant leurs environnements d'exécution d'origine. *SAM-RT* est donc un SOA abstrait d'où son nom : *Service Abstract Machine RunTime*.

Dans le chapitre introduction (cf. Chapitre 1 - Figure 2), nous avons vu que les technologies SOA peuvent avoir des buts et des comportements différents. En effet dans le cas d'un SOA – donc d'une technologie spécifique – les informations de l'environnement et celle du comportement des services sont connues. Prenons par exemple le cas de la disponibilité d'un service, cette disponibilité est inhérente au SOA. La technologie UPnP est utilisée dans l'informatique ubiquitaire car le protocole qu'il fournit permet de découvrir dynamiquement les services présents dans le contexte d'exécution (ici un réseau local) d'un client. Un client accepte implicitement le fait que les services qu'il utilise peuvent partir à tout moment. Les Web-Services, eux, sont utilisés dans les infrastructures intra ou inter entreprise car cette technologie permet l'interopérabilité et donc facilite l'intégration des systèmes entre eux. Cependant, dans le cas des Web-Services, la dynamicité des services n'est pas un requis, loin de là : le service est un élément statique du système. Les développeurs des clients de Web Services ne cherchent pas à gérer la dynamicité car quand on utilise cette technologie : on assume implicitement la staticité du service.

Mais dans le cas de SAM, on prend en charge les différents SOA existants tout en masquant leurs technologies. La connaissance de la technologie et donc les informations implicites ne doivent pas être perdues. Par conséquent *SAM-RT* prédéfinit un certain nombre de propriétés caractérisant en partie la nature et le comportement dynamique de chaque concept. Or le méta-modèle de *SAM CORE* défini dans la section 2 de ce chapitre modélise le concept de Service dans l'approche SAM : il ne définit ni le SOA, ni le caractère dynamique du contexte d'exécution, ni les opérations (sa projection) sur chacun de ces concepts. Cette section spécialise le méta-modèle *SAM CORE* dans le but d'une part de modéliser l'état d'exécution des services et d'autres part d'abstraire les opérations d'administration nécessaire à l'adaptation. En résumé, le méta-modèle de cette section définit à la fois un modèle qui est à la fois **descriptif** et **prescriptif**.

Dans cette section, nous allons définir dans un premier temps la projection des concepts de *SAM CORE* à l'exécution ; chaque concept sera caractérisé par un ensemble d'opérations possibles et par un ensemble de propriétés caractérisant sa nature et son comportement dynamique. Nous définirons ensuite un ensemble de mécanismes permettant la publication, la sélection et l'invocation (d'instance) de service nécessaires à tous SOA. Les problématiques liées à l'abstraction des technologies seront traitées dans la section 4 de ce chapitre.

3.1 OPERATIONS ET CARACTERISATIONS

Nous constatons au travers de la logique de groupe d'équivalence que le méta-modèle *SAM CORE* fournit les relations suivantes :

- Une spécification est le type d'un ensemble d'implémentations ;
- Une implémentation est une instance de spécification mais aussi le type pour un ensemble d'instances de services ;
- Une instance est une instance d'une implémentation

En d'autres termes les représentants sont types et instances à la fois. Les trois matérialisations d'un service sont toutes des instances qui coexistent dans l'environnement d'exécution. Bien qu'elles aient

des relations classes/objet, le méta-modèle SAM CORE les définit dans un même modèle. Le méta-modèle SAM CORE peut être utilisé pour deux buts :

- Dans un but descriptif où le modèle obtenu représente l'état d'exécution en termes de service ;
- Dans un but prescriptif où le modèle est utilisé pour agir sur l'environnement d'exécution.

Dans *SAM-RT*, le méta-modèle SAM CORE est utilisé pour définir un modèle descriptif homogène de l'état d'exécution des services dans l'environnement d'exécution, masquant ainsi la complexité et les problématiques liées à l'utilisation de technologies à service hétérogènes. Le modèle ainsi obtenu permet aussi de manipuler les différentes matérialisations.

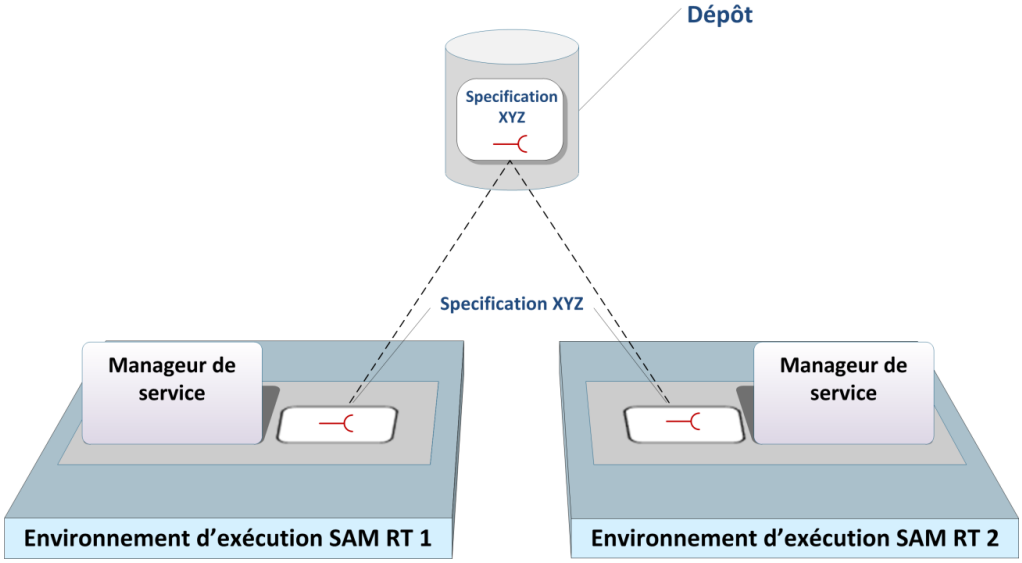


Figure 46 Cohérence dans l'environnement SAM

Un certain nombre d'opérations de manipulation sont interdites dans l'environnement d'exécution. En effet modifier des caractéristiques d'un élément pose un certain nombre de problème. Dans l'exemple ci-dessus, si l'on supprime une dépendance à la spécification XYZ sur SAM-RT 1, la spécification résultante est-elle toujours XYZ, faut-il modifier celle de SAM-RT 2 ? Le problème soulevé par cet exemple est celui de l'évolution (versionnement, révision…) d'un élément logiciel (cf. [LevT10]). Nous ne souhaitons pas dans cette thèse traiter du versionnement ou de la révision dynamique à l'exécution. Nous n'autorisons donc que trois manipulations :

- La création d'un membre à partir de son représentant (modulo les propriétés de configuration) ;
- La destruction d'un membre ;
- L'ajout, la modification et la suppression de propriétés.

Les opérations sur les propriétés sont cependant limitées aux aspects suivants :

- un membre ne peut pas supprimer une propriété héritée de son représentant ni même la modifier ;
- une propriété variable d'un membre n'est que modifiable, jamais supprimable ;
- certaines propriétés variables peuvent seulement être définies à la création du membre (propriété de configuration) et ne peuvent donc pas être modifiées.

En résumé, nous autorisons seulement les opérations permettant de modifier l'architecture de l'état d'exécution (opérations de manipulation) et l'enrichissement sémantique des concepts à l'exécution (ajout de propriétés).

Dans la suite de cette partie nous définirons donc quels sont respectivement les opérations de manipulation et les opérations d'introspection que l'on a sur chaque concept, ainsi que les propriétés prédéfinies – le type de la valeur sera précisé ainsi que son optionalité.

3.1.1 Spécification

Il faut bien comprendre que dans SAM, une spécification a une existence à l'exécution et n'est pas dérivé d'un autre concept. Prenons par exemple le cas d'OSGi : une instance de service définit ses fonctionnalités au travers des interfaces qu'elle implémente et qui sont enregistrées dans le registre. Pour autant, de notre point de vue, une interface Java n'est pas une spécification de service **dans OSGi : une interface Java définit un service que s'il est dans le registre de service**. La spécification est donc dérivée de l'instance et n'a donc pas d'existence propre, son cycle étant lié à celui de l'instance. **Dans notre cas, une spécification est un élément logiciel à l'exécution** qu'elle est ou non des implémentations et des instances en exécution.

Rappelons qu'une *spécification* est constituée d'un ensemble d'interfaces, de propriétés et de dépendances. Elle a pour but de spécifier un (composant orienté) service.

Introspection

- ***getID*** : renvoie l'identificateur de la spécification ;
- ***getName*** : renvoie le nom de la spécification (nom symbolique) ;
- ***getImplementations*** : cette opération renvoie l'ensemble des implémentations de cette spécification présentes dans l'environnement d'exécution ;
- ***getDependencies*** : renvoie les références de dépendance (voir section 3.1.4 de ce chapitre) ;
- ***getInterfaces*** : renvoie les interfaces invocables définies par la spécification ;
- ***getProperties*** : renvoie les propriétés *prédéfinies* et *propres* de la spécification.

L'ensemble de ces opérations permet d'avoir accès à l'ensemble des caractéristiques d'une spécification.

Manipulation

Comme expliqué précédemment, les opérations de manipulation sont limitées. Une spécification est le représentant de ses implémentations et est aussi membre d'un groupe de plus haut niveau (non défini volontairement). Elle peut donc avoir des propriétés propres.

- ***setProperty*** : permet d'ajouter ou modifier une propriété propre à la spécification ;
- ***removeProperty*** : permet de supprimer une propriété propre ;
- ***createImplementation(parameters)*** : comme tout représentant, une spécification peut instancier / générer des implémentations ; prenons par exemple les *Dynamic Proxy*[12] définis par Java, l'implémentation (le proxy) est générée à partir des interfaces et où le code fonctionnel est un paramètre d'instanciation. Cette opération permet donc d'instancier une implémentation si celle-ci est instanciable.

Propriétés prédéfinies

Les propriétés prédéfinies sont des propriétés variables des groupes d'équivalence dont la sémantique et la syntaxe sont connues.

- ***instanciable* [*boolean*]** : cette propriété est vraie si la spécification est instanciable, fausse sinon ;

[12] http://download-lnw.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/Proxy.html

- ***singleton*** **[*boolean*]** : cette propriété est vraie s'il n'est possible d'instancier qu'une seule implémentation, ou s'il ne peut y avoir qu'une seule implémentation dans l'environnement d'exécution, fausse sinon ;
- ***sharable*** **[*boolean*]** : cette propriété est vraie si la spécification peut être utilisée par plusieurs consommateurs de service simultanément, fausse sinon. Son utilité sera expliquée dans la section 3.2 de ce chapitre.

Dans notre cas, la caractérisation du comportement dynamique d'une spécification se résume à sa disponibilité (cf. Chapitre 2 - section 2.1.1).

- ***availability*** **[*dynamic*|*static*]** : cette propriété a pour valeur ***dynamic*** si la spécification a une disponibilité dynamique ou semi-dynamique, et donc ***static*** si elle a une disponibilité statique ou semi-statique. La distinction entre dynamique et semi-dynamique ou statique et semi-statique se fait via les autres propriétés. En effet il n'est pas possible d'instancier ou de détruire un service dans le cas d'une disponibilité purement dynamique ou purement statique : dans un cas l'environnement d'exécution n'a pas accès au cycle de vie du service, et dans l'autre : le service est constamment présent durant le cycle de vie de l'environnement d'exécution et donc il ne peut être ni instancié ni détruit.

3.1.2 Implémentation

Une *implémentation* implémente une spécification et est constituée d'un ensemble de propriétés et de dépendance. Elle a pour but d'implémenter un service.

Introspection

- ***getID*** : renvoie l'identificateur de l'implémentation ;
- ***getName*** : renvoie le nom de l'implémentation (nom symbolique) ;
- ***getSpecification*** : renvoie la spécification qu'implémente l'implémentation ;
- ***getInstances*** : cette opération renvoie l'ensemble des instances de cette implémentation présent dans l'environnement d'exécution ;
- ***getDependencies*** : Contrairement aux interfaces, une implémentation peut avoir plus de dépendance que sa spécification. Par conséquent cette opération renvoie les références de dépendance de l'implémentation (voir section 3.1.4 de ce chapitre) et pas uniquement celle de sa spécification ;
- ***getProperties*** : renvoie les propriétés prédéfinies et propres de l'implémentation.

Il n'y a pas d'accesseur direct aux interfaces, car celle-ci sont celles de sa spécification.

L'ensemble de ces opérations permet d'avoir accès à l'ensemble des caractéristiques d'une implémentation.

Manipulation

- ***setProperty*** : permet d'ajouter une propriété propre ou de modifier une propriété propre ou variable ;
- ***removeProperty*** : permet de supprimer une propriété propre ;
- ***createInstance(parameters)*** : permet d'instancier une instance de service si l'implémentation est instanciable. Par définition une implémentation sert à créer des instances ; cependant nous verrons dans la section 3.2 de ce chapitre qu'il n'est pas toujours possible d'instancier une implémentation ;
- ***dispose*** : cette opération permet de disposer/détruire l'implémentation si celle-ci est disposable.

Propriétés prédéfinies

- ***instanciable*** **[*boolean*]** : cette propriété est vraie si l'implémentation est instanciable, fausse sinon ;
- ***singleton*** **[*boolean*]** : cette propriété est vraie s'il n'est possible d'instancier qu'un seul membre, ou s'il ne peut y avoir qu'un seul membre dans l'environnement d'exécution, fausse sinon ;
- ***disposable*** **[*boolean*]** : cette propriété est vraie si l'implémentation peut être disposée/détruite, fausse sinon ;
- ***sharable*** **[*boolean*]** : cette propriété est vraie si l'implémentation peut être utilisée par plusieurs consommateurs de service simultanément, fausse sinon. Son utilité sera expliquée dans la section 3.2 de ce chapitre.

Dans notre cas, la caractérisation du comportement dynamique d'une implémentation se résume à sa disponibilité (cf. Chapitre 2 - section 2.1.1).

- ***availability*** **[*dynamic*|*static*]** : cette propriété a pour valeur ***dynamic*** si l'implémentation a une disponibilité dynamique ou semi-dynamique, et ***static*** si elle a une disponibilité statique ou semi-statique. Pour les mêmes raisons que pour la spécification, la distinction entre dynamique et semi-dynamique ou statique et semi-statique se fait via les autres propriétés.

3.1.3 Instance

Une *instance* est une instance configurée d'implémentation ; elle est constituée d'un ensemble de propriétés et de dépendances. L'instance est l'élément fonctionnel d'un service ; c'est-à-dire qu'elle fournit les opérations fonctionnelles du service.

Introspection

- ***getID*** : renvoie l'identificateur de l'instance ;
- ***getName*** : renvoie le nom de l'instance (nom symbolique) ;
- ***getImplementation*** : renvoie l'implémentation de l'instance ;
- ***getDependencies**** : renvoie les références de la dépendance (voir section 3.1.4 de ce chapitre) ;
- ***getProperties*** : renvoie les propriétés *prédéfinies* et *propres* de l'instance ;
- ***getServiceObject***** : renvoie l'objet qui représente le service.

(*) Bien qu'ayant exactement les mêmes références de dépendance de son implémentation, une instance dépend à l'exécution d'une autre instance. Voir la section 3.1.4 de ce chapitre.

(**)Cette opération n'a pas de sens si l'instance SAM est considérée comme la représentation (un modèle) d'une « vraie » instance; elle a un sens dans le cas où SAM est considérée comme un SOA à part entière, cette opération permet d'invoquer effectivement le service (cf. section 3.2 de ce chapitre).

Manipulation

- ***setProperty*** : permet d'ajouter une propriété propre ou de modifier une propriété propre ou variable ;
- ***removeProperty*** : permet de supprimer une propriété propre ;
- ***dispose*** : cette opération permet de disposer/détruire l'instance si celle-ci est disposable ;
- ***invoke***** : cette opération permet d'invoquer une fonctionnalité de l'instance du service. Cette opération est similaire à de la réflexion, car celle-ci correspond à l'invocation de l'objet du service (*Service Object*).

Propriétés prédéfinies

Toutes les propriétés prédéfinies d'une instance sont des propriétés variables ou communes définies par leur représentant (implémentation).

- ***disposable* [*boolean*]** : cette propriété est vraie si l'instance peut être disposée/détruite, fausse sinon ;
- ***sharable* [*boolean*]** : cette propriété est vraie si l'instance peut être utilisée par plusieurs consommateurs de service simultanément, fausse sinon. Son utilité sera expliquée dans la section 3.2 de ce chapitre;

Dans notre cas, la caractérisation du comportement dynamique d'une instance se résume à sa disponibilité (cf. Chapitre 2 - section 2.1.1).

- ***availability* [*dynamic*|*static*]** : cette propriété a pour valeur ***dynamic*** si l'instance a une disponibilité dynamique ou semi-dynamique, et ***static*** si elle a une disponibilité statique ou semi-statique. Pour les mêmes raisons que pour la spécification, la distinction entre dynamique et semi-dynamique ou statique et semi-statique se fait via les autres propriétés.

3.1.4 Dépendance

La projection du concept de *Dépendance* diffère du concept défini dans SAM CORE. En effet le méta-modèle SAM CORE est une vue « Top-Down » : de la conception à l'exécution. Par conséquent à la conception, les spécifications dépendent entre elles, puis on les raffine en implémentation. Lors du développement une implémentation peut être liée pour des raisons techniques à l'utilisation d'une implémentation précise voir même d'une configuration d'une implémentation précise. Par conséquent les dépendances inhérentes à l'implémentation n'est plus seulement la spécification mais un raffinement de cette spécification comme le montre la Figure 47.

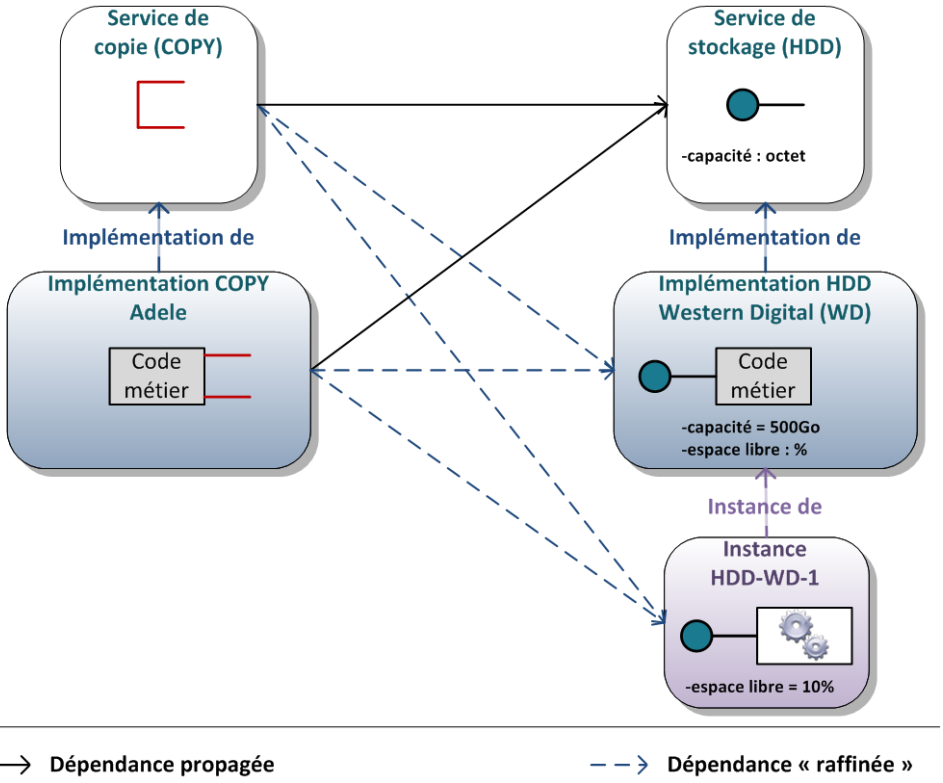


Figure 47 Dépendances dérivées

Cependant à l'exécution, techniquement, une instance ne dépend pas d'une spécification mais bien d'une autre instance. En d'autres termes il y a la définition de la dépendance et le service en cours d'utilisation. Dans notre cas, une dépendance fournit à la fois la définition est l'état à l'exécution de cette dépendance.

Dans notre cas, la définition d'une dépendance de service se fait via un filtre.

Introspection

- ***getID*** : renvoie l'identificateur de la dépendance ;
- ***getRequired*** : renvoie le ou les services ciblés par la définition de la dépendance ;
- ***getUses*** : renvoie le ou les services en cours d'utilisation ;
- ***getProperties*** : renvoie les propriétés prédéfinies et propres de la dépendance ;
- ***getFilter*** : renvoie le filtre utilisé pour la sélection du ou des services.

Manipulation

- ***setProperty*** : permet d'ajouter une propriété ;
- ***removeProperty*** : permet de supprimer une propriété ;
- ***setFilter*** : la modification du filtre entraîne la réévaluation immédiate de la dépendance, cette opération n'est acceptée que :
    - si la dépendance est substituable ;
    - si le service déjà sélectionné n'a pas encore été utilisé et donc n'est pas encore lié.

Il est à noter que la modification du filtre à l'exécution est rarement implémentée dans les approches orientées service car celle-ci nécessite une réévaluation de la sélection pouvant invalider la sélection courante. Pour cela il faut que le consommateur de service autorise la substitution ou que le service sélectionné n'ait pas encore été utilisé.

En théorie, un service se comporte comme n'ayant pas d'état ; cela permet le partage et la substitution à tout moment. Mais en pratique les services ne sont pas sans état limitant ainsi la substitution qu'à des cas bien précis.

Propriétés prédéfinies

Une dépendance peut cibler zéro ou plusieurs services à la fois. La propriété de cardinalité permet de définir le nombre d'éléments ciblés par la définition de la dépendance. La valeur de la cardinalité permet d'induire des propriétés intrinsèques de la dépendance ; par exemple si la cardinalité est 0-1 alors cela signifie que la dépendance requiert **optionnellement** un service. Et si la cardinalité est à * alors cela signifie que la dépendance est **multiple**.

- ***Cardinality* [*0-1*| *1* | *1-** | *]** : cette propriété indique le nombre les services ciblés par la définition de la dépendance et non le nombre de services en cours d'uilisation.

L'opération de manipulation *setFilter* dépend de la tolérance de la liaison à la disponibilité dynamique. Notons que cette tolérance est inhérente au service qui la référence ; en effet un service n'utilise pas n'importe quel service, mais l'utilise via une interface : il connait donc sa sémantique. De plus, la dépendance ne tolère pas nécessairement tous les aspects de la disponibilité dynamique. Nous allons prendre différents exemples pour montrer différentes propriétés des dépendances.

Une dépendance multiple n'est pas nécessairement optionnelle ; elle ne supporte pas non plus nécessairement l'ajout ou la suppression dynamique. En effet, le service peut requérir à son initialisation des services, qui une fois utilisés ne peuvent disparaitre sans causer une faute chez le consommateur de service.

Prenons le cas de JORAM[13] une implémentation de la spécification JMS [JMS99], l'ajout dynamique de serveur de message dans un cluster est possible, chacun des serveurs connaissant la liste des autres. Cependant pour des raisons de persistance de message il n'est pas possible de supprimer dynamiquement un serveur.

- ***Addable* [*boolean*]** : cette propriété est vraie si la découverte et l'utilisation d'un service (cf. Figure 48 Méta-modèle SAM CORE à l'exécution) peut se faire à tous moments, fausse sinon ;
- ***Removable* [*boolean*]** : cette propriété est vraie si la disparition, à tous moments, d'un service utilisé est supportée (cf. Figure 48 Méta-modèle SAM CORE à l'exécution), fausse sinon ;
- ***ModifiableFilter* [*boolean*]** : cette propriété est vraie si la modification et la réévaluation des services utilisés du filtre sont autorisées à tout moment, fausse sinon.

Bien qu'elles n'aient pas toutes un sens, la combinaison de ces trois propriétés permet de définir l'ensemble des scénarii possibles définissant le comportement dynamique autorisée par une dépendance. Par exemple la propriété ***Substituable*** est équivalente à (*cardinality*=1) ^ *addable* ^ *removable*. La dépendance d'un serveur de message JORAM vers les serveurs du cluster se caractérise de la manière suivante : (*cardinality*=1-*), *(addable=true)* et *(removable=false)*.

Tant qu'un service sélectionné n'est pas utilisé (chargé / référencé par le code fonctionnel), il est toujours possible de modifier le filtre et/ou de substituer les services sélectionnés.

3.1.5 Méta-modèle SAM CORE à l'exécution

Le méta-modèle *SAM CORE* a pour but de définir des concepts communs durant le cycle de vie d'une application orientée service. Cependant ce méta-modèle ne définit ni l'utilisation ni la signification d'un concept aux différentes phases du cycle de vie.

La Figure 48 est la projection du méta-modèle SAM-CORE à l'exécution ; elle prend en compte les différents aspects (opérations et propriétés) définis dans les sections précédentes. Notons que le méta-modèle obtenu n'est pas un raffinement de celui d'origine ; en effet le méta-modèle d'origine contraint l'utilisation des dépendances ; or ici la notion de dépendance est au niveau service (donc plus générique que le méta-modèle de SAM CORE d'origine).

[13] *Java™ Open Reliable Asynchronous Messaging* : http://joram.ow2.org/technical.html

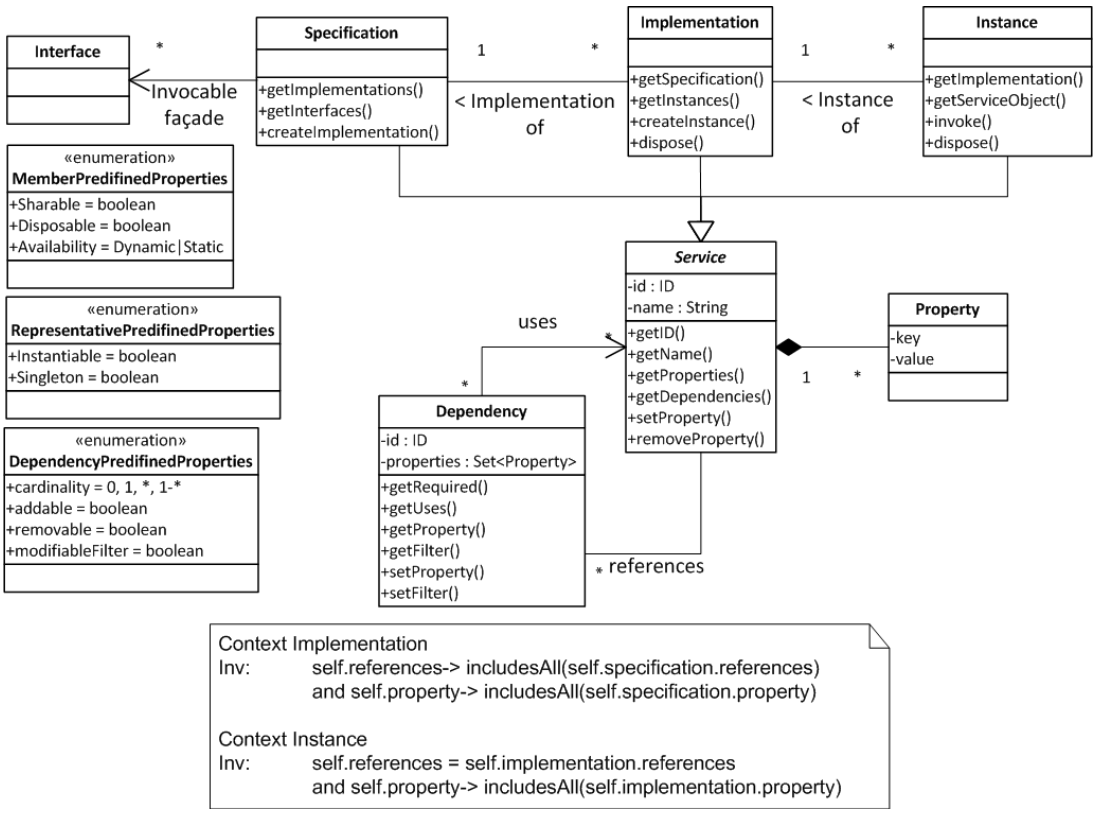


Figure 48 Méta-modèle SAM CORE à l'exécution

3.2 SOA SAM

L'environnement d'exécution *SAM-RT* est un SOA abstrait ; il possède donc les mécanismes de publication, de sélection et de communication propres à l'approche orientée service.

Registre de service

Contrairement à la plupart des autres approches, la SAM exhibe les trois matérialisations d'un service. Contrairement aux autres SOA, le registre de services n'est pas uniquement un registre d'instances mais un ensemble de trois registres que sont les registres de spécification, d'implémentation et d'instance.

Sélection

SAM CORE est un méta-modèle composant orienté service dont toutes les matérialisations de service sont distinctes et identifiées. La sélection d'une instance de service peut se faire, d'une part comme dans tous les SOA, à l'aide d'un filtre ; d'autre part à l'aide d'un identificateur. Bien qu'utiliser un identificateur pour sélectionner une instance de service ne soit pas dans la logique de l'approche orientée service, il est en pratique couramment utilisé dans la définition d'application (cf. section 2.3.4 de ce chapitre).

En principe lorsqu'on utilise un filtre pour sélectionner un service, on ne connait pas le service qui va être retourné ; il se peut que plusieurs services correspondent aux critères de sélection ou aucun. Par identification le résultat est prédictif, soit le service correspondant est présent sur l'environnement soit il ne l'est pas.

Il est intéressant de remarquer qu'il est possible de sélectionner un service en navigant à travers les liens du modèle. Prenons l'exemple suivant (cf. Figure 49) : un consommateur désire un service défini par la spécification COPY. Pour cela il utilise le registre de spécification. Il recherche à partir de la spécification l'ensemble des implémentations, puis finalement des instances de cette implémentation ; pour finalement l'utiliser.

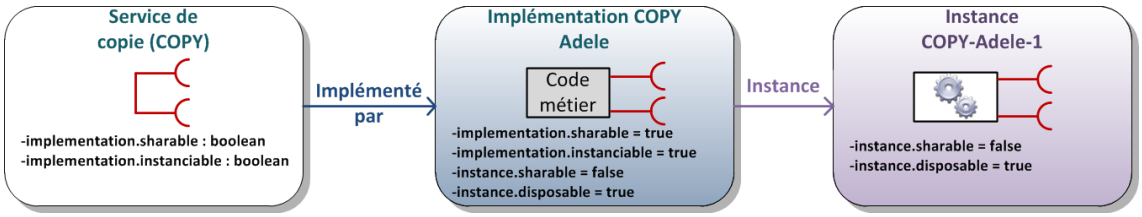


Figure 49 Exemple de sélection par navigation

La sélection précédente aurait pu tout simplement être réalisée de la manière suivante : le consommateur de service sélectionne une instance de service dans le registre d'instance avec la spécification indiquée dans le filtre.

Dans notre exemple, l'implémentation a défini « sharable = false » et « disposable = true » ce qui signifie que les instances de cette implémentation ne peuvent pas être partagées par deux consommateurs, et elles peuvent être détruites. Si l'instance est utilisée par un autre service, alors le consommateur peut instancier sa propre instance de service, et la détruire à la fin de son utilisation.

Notification

La section précédente a défini un certain nombre d'opérations de manipulation de services. Ces manipulations peuvent être faites à tout moment, or la création, la destruction ou la modification de propriétés peuvent impacter un certain nombre de consommateurs de service. Par conséquent, l'environnement d'exécution SAM-RT définit un mécanisme de notification basé sur le patron « publie/souscrit » (ou en anglais : *publish / subscribe pattern*).

Chaque registre émet des évènements (**publieur**) de trois types :

- la disponibilité d'un service ;
- l'indisponibilité d'un service ;
- la modification des propriétés d'un service.

Un évènement désigne l'identificateur du service correspondant. Les registres (**publieur**) ne font pas de distinction entre la disponibilité dynamique, semi-dynamique ou semi-statique (cf. Chapitre 2 - section 2.1.1). Par conséquent, ils envoient un évènement à tous les **souscripteurs** même si l'un d'entre eux est à l'origine de ce changement (création, destruction ou modification). Tout service peut souscrire à un ou plusieurs registres via un ou des *handlers* (éléments de rappel).

3.3 SYNTHESE

Le méta-modèle de service *SAM CORE* définit dans la section 2 de ce chapitre est une vision abstraite pour la conception et ne correspond pas aux besoins d'un environnement d'exécution. Ce méta-modèle est réduit au concept de service, il ne définit pas les concepts d'application, d'unité de déploiement,… Dans cette thèse, ce méta-modèle a pour but de définir des modèles de l'état de l'architecture des services à l'exécution.

Le méta-modèle SAM CORE est basé sur une approche **composant orienté service** (cf. Chapitre 2 - section 4.4). Ce méta-modèle permet d'exhiber les concepts d'instance et d'implémentation. Avoir les concepts d'instance et de dépendance permet de définir une architecture, ce qui dans notre cas permet

de décrire l'état d'exécution d'une plate-forme à service par un **modèle descriptif** de service. Avoir le concept d'implémentation permet d'agir sur le cycle de vie d'une instance de service car à partir d'une implémentation il est possible d'instancier, de supprimer, … en d'autres termes avoir le concept d'implémentation permet de définir les opérations de manipulation de l'architecture. De ce point de vue le **modèle** d'état devient **prescriptif** car permet la modification de l'architecture.

SAM-RT fournit un registre pour chaque matérialisation de service : spécification, implémentation et instance. A partir d'un de ces registres, il est possible d'accéder au modèle d'état d'exécution des services. Une instance de service peut donc être sélectionnée via le registre d'instance ou être instancier et sélectionner à partir d'une implémentation obtenue par navigation dans le modèle à partir du registre de spécification. Finalement SAM RT fournit un mécanisme de notification pour signaler les changements d'état (apparition ou disparition) des différentes matérialisations. En résumé SAM-RT fournit un ensemble de registre permettant la sélection d'un service ainsi qu'un mécanisme de notification de la disponibilité dynamique des services. SAM-RT possèdent toutes les propriétés d'un **SOA dynamique**.

La Figure 50 définit le méta-modèle de SAM-RT basé sur la projection de SAM CORE correspondant au besoin de l'exécution.

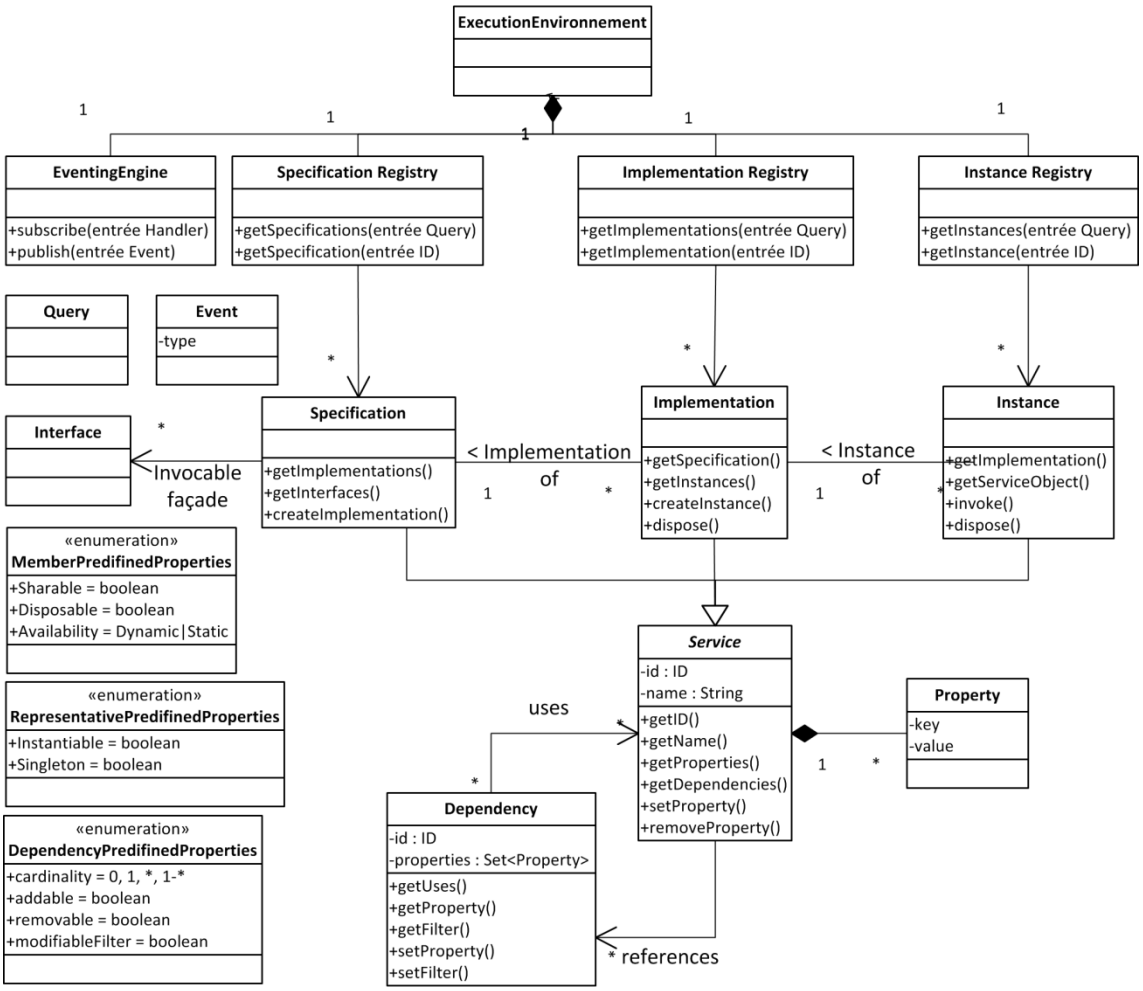


Figure 50 Méta-modèle de *SAM-RT*

4. ENVIRONNEMENT DISTRIBUE HOMOGENE

L'objectif de l'approche SAM est de définir et d'exécuter des applications pouvant être adaptées à l'exécution telles que l'exemple explicité dans l'introduction de ce chapitre (cf. Figure 1). La réalité du contexte d'exécution des applications est de nos jours un environnement distribué. On fédère les systèmes via des ESB [Cha04] ou des EAI [Lin00] – qui peuvent se résumer comme étant un ensemble de ponts de systèmes spécifiques vers un système commun ; ces intégrations de systèmes ne se font pas sans mal. Pour exprimer le problème sous-jacent, nous allons prendre l'exemple de la Figure 51 : l'OSGi Alliance a standardisé [OSGib] – pour les plates-formes OSGi – un pont vers la technologie UPnP sous le nom de : *UPnP Basedriver*. Ce pont permet de réifier les équipements UPnP sur le réseau dans l' « espace d'exécution » (cf. Annexe A -) OSGi et inversement de pouvoir exporter des services OSGi en tant qu'équipement UPnP.

Imaginons que le service A d'OSGi soit un proxy requérant un service ayant la même interface. Si l'UPnP *Basedriver* est mal implanté, alors potentiellement celui-ci peut importer le service A qui avait été exporté, créant ainsi une boucle d'appel infinie. Bien évidemment la spécification de l'UPnP *Basedriver* explicite le fait qu'un service exporté ne doit pas être importé. Le *Basedriver* créé une image du service A, il est donc capable d'identifier l'image du service comme étant celle du service A.

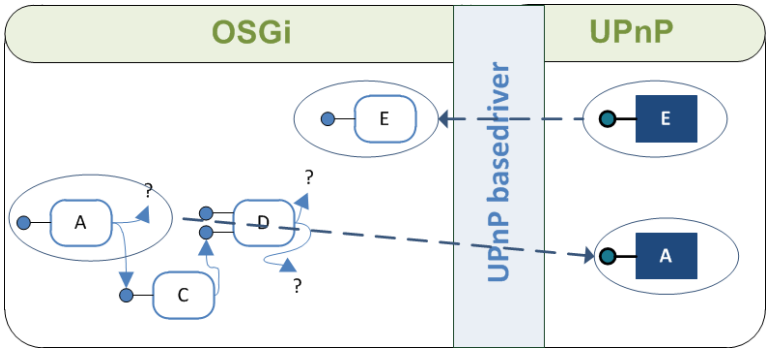


Figure 51 Espaces d'exécution : pont

Prenons maintenant l'exemple de la Figure 52 : il y a trois espaces d'exécutions : OSGi [OSGi-S], DPWS [DPWS-S] et UPnP [UPnP] [UPnP-S]. Chaque espace est relié aux autres par un pont :

- entre OSGi et UPnP via l'UPnP Basedriver ;
- entre OSGi et DPWS via le DPWS Basedriver [OSGic] ;
- entre UPnP et DPWS via un pont X.

Dans cet exemple le service A de la plate-forme OSGi est exporté d'un part vers UPnP via l'UPnP basedriver, et d'autre part vers DPWS via le DPWS basedriver. A partir de la plateforme d'OSGi il est possible de connaitre et de lier les trois identifiants (OSGi, UPnP et DPWS). Cependant le pont X n'a pas connaissance de ce lien et par conséquent les services UPnP et DPWS de A sont, pour lui, des services disjoints. Puis que pour le pont X il n'y a pas de lien entre les services A d'UPnP et DPWS, ils les exportent réciproquement en service A'. Et finalement les services A' d'UPnP et DPWS sont importés dans la plate-forme OSGi. Au final au bout d'un tour de boucle, la plate-forme d'exécution OSGi possède trois services distincts : un service source et deux images. Bien que le pont X connaisse la relation entre les identifiants d'UPnP et DPWS, OSGi les ignorent et donc il n'est pas possible d'établir la relation à partir d'OSGi. En résumé, l'identification entre ces trois espaces sont hétérogènes, seul le pont est capable de faire le lien entre un service source et son image ; par conséquent cette image peut être à son tour reprise par un autre pont qui va faire une image, etc, etc.

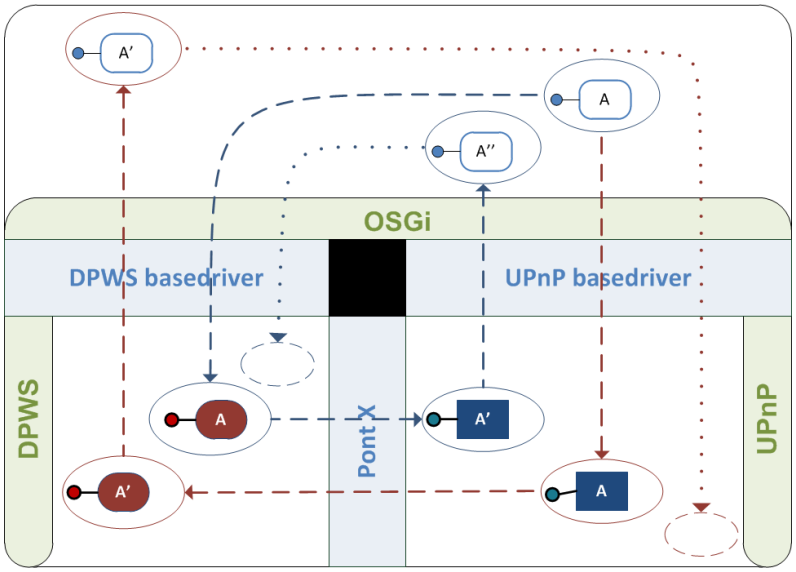


Figure 52 Boucle de réification

Cet exemple soulève le problème de l'hétérogénéité de l'identification entre les espaces d'exécution qui est une conséquence de l'hétérogénéité et du cloisonnement des environnements d'exécution ; chacun a son protocole de nommage, de communication, de formatage … tout un ensemble d'informations qui ne sont pas toujours partagées causant potentiellement des boucles, des duplications, des conflits…

4.1 ENVIRONNEMENT HOMOGENE

Le problème d'identification soulevé précédemment peut être résolu de deux façons :

- soit le contexte d'exécution est fermé, contrôlé et restreint ; ce qui nous permet de définir les espaces d'exécution et leurs interactions. C'est-à-dire que nous avons la parfaite connaissance des différents environnements du système, et que nous pouvons garantir qu'il n'y aura pas de conflit.
- soit la *connaissance* des ponts en termes d'identification est partagée ; ce qui permet d'une part de mettre en place des algorithmes équivalents au « *spanning tree* » ; et d'autre part permet d'identifier l'espace d'exécution source. Identifier la source peut permettre à un client DPWS d'interagir avec le service A plutôt qu'avec le service A' de son espace d'exécution.

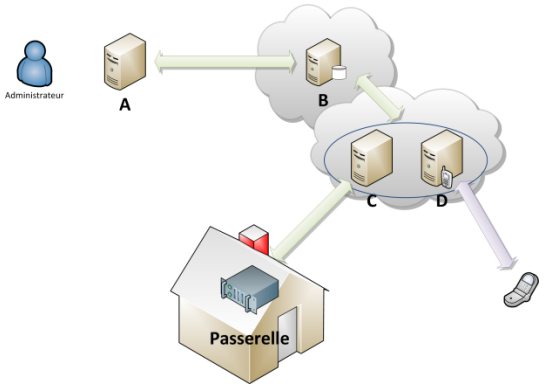


Figure 53 Environnement d'exécution distribué

L'approche SAM milite clairement pour la seconde solution, nous voulons obtenir un contexte d'exécution homogène où chaque élément est distinctement identifiable.

Simplifions l'exemple défini dans l'introduction par la Figure 53 : les administrateurs de l'entreprise XYZ accèdent à partir de leur machine (A) au système via le serveur (B) qui gère à la fois les médias et les abonnements des clients. L'entreprise met en place deux serveurs : le serveur (C) qui gère les passerelles multimédia et le serveur (D) qui gère les équipements nomades tels que les smartphones ou les ordinateurs portables 3G.

Les éléments de l'infrastructure ont des rôles et des besoins différents et donc des technologies différentes comme schématisé dans la Figure 54.

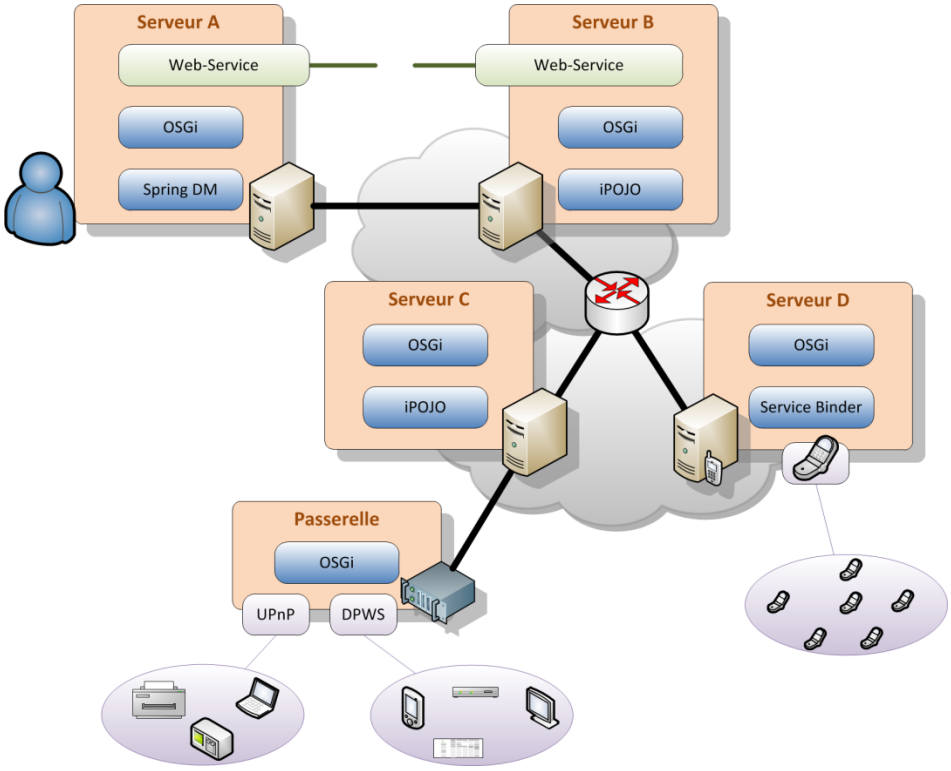


Figure 54 Exemple de répartition des technologies dans une infrastructure distribuée

Par conséquent, l'environnement d'exécution *SAM-RT* est construit au-dessus d'une représentation de machine nommée : *Abstract Machine*.

4.2 ABSTRACT MACHINE

L'*Abstract Machine* (AM) n'est pas en soi un environnement d'exécution, elle ne fournit pas nativement d'infrastructure pour l'exécution d'applications telles qu'un SOA ou un orchestrateur. Ces fonctionnalités sont limitées à la gestion de la distribution et à la gestion de l'environnement d'exécution.

D'un côté elle permet la connexion vers zéro ou plusieurs autres AM, et de l'autre elle permet de spécialiser cette machine par l'ajout d'une ou plusieurs extensions.

4.2.1 Distribution

Quasiment tous les protocoles de communication sont basés sur l'identification des éléments qui constituent le réseau ; c'est-à-dire l'identification des émetteurs/récepteurs. La raison est triviale : un émetteur envoie un message à un récepteur qui potentiellement peut lui répondre ; plus simplement s'il n'y a pas d'adresse (identificateur) sur le courrier, le facteur ne peut pas deviner le destinataire. De plus, comme pour les services, une machine est utilisée dans un but qui dépend de ses propriétés, que ce soit sa localisation ou ses ressources. Une *Abstract Machine* a donc un identificateur.

Nativement, chaque AM peut se connecter à une autre AM via son identificateur (s'il existe bien évidemment un moyen de communication entre les deux). Chaque machine connectée est réifiée dans un registre de machine propre à chaque AM. La mise en place d'un registre de machine permet :

- d'accéder aux fonctionnalités (extensions) d'une machine distante par des extensions de l'AM ;
- De découpler la découverte des AM du noyau et donc de pouvoir utiliser un ou plusieurs protocoles de découverte adaptés au contexte de la plate-forme.

Le but sous-jacent à l'AM est de fournir le noyau minimum et générique pour tous les types d'environnements. Par conséquent, l'AM n'impose pas de topologies réseaux particulière. Chaque AM **peut être connecté à un ou plusieurs réseaux** ; par exemple une AM peut avoir une carte réseau wifi et une carte réseau avec des câbles RJ45. La Figure 55 contient 4 exemples de topologies possibles entre cinq AM.

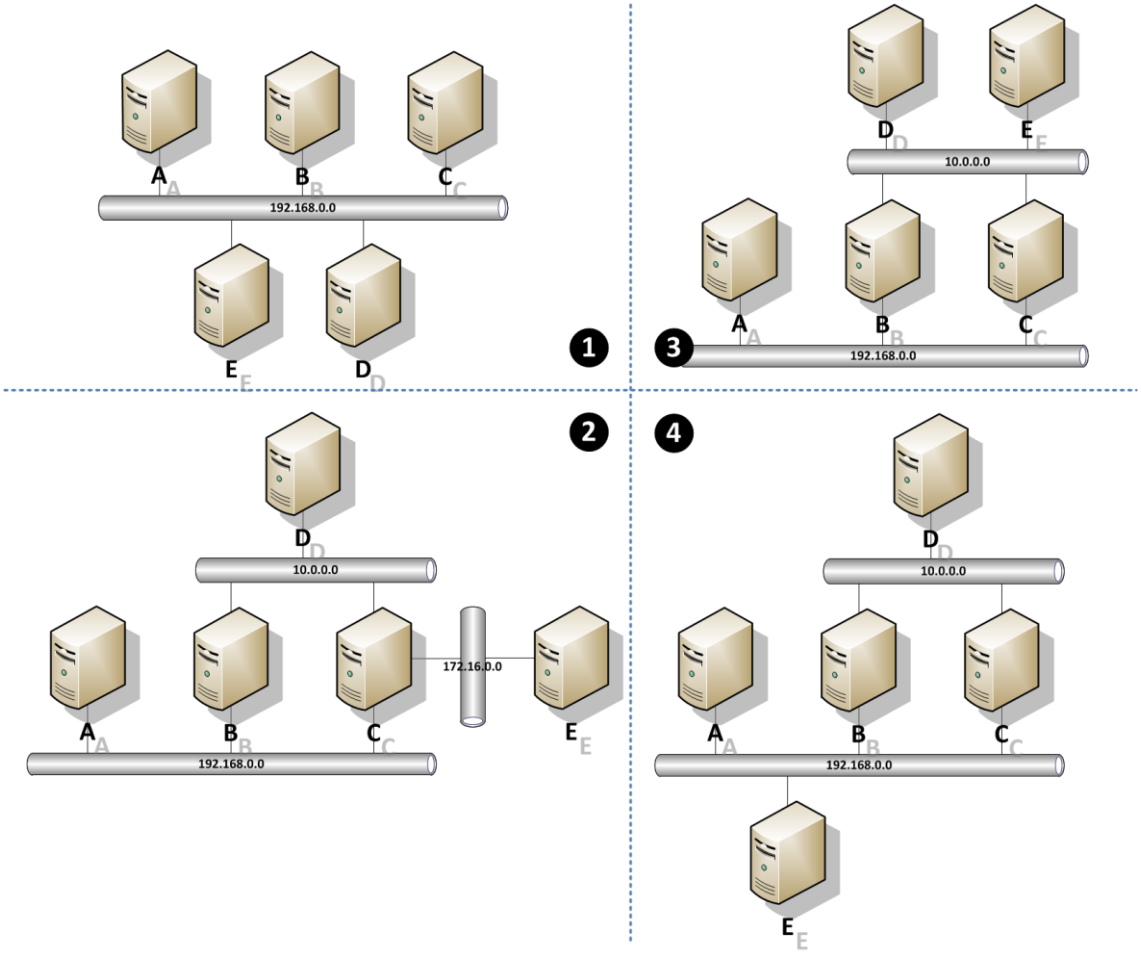


Figure 55 Exemple de topologies

Il faut bien voir que la topologie du réseau impacte l'accessibilité des AM et par extension la découverte. En effet, pourquoi vouloir découvrir une machine si je ne peux pas y accéder. Par exemple dans les schémas 2 et 3 l'AM A ne peut accéder qu'à B et C. Une AM n'a pas vocation à être nativement un routeur.

Par la suite l'ensemble des AM interconnectées sera désigné par AM*.

Découverte

Une AM fournit nativement une API permettant la connexion à d'autres AM, mais ne fournit pas par défaut de protocoles de découverte : ceux-ci sont externalisés du noyau. Chaque AM peut avoir zéro ou plusieurs protocoles de découverte. Les protocoles de découvertes d'AM basés sur des technologies comme *WS-Discovery* [WSD09], *SSDP* [SSDP99] ou *Apple Bonjour* [Bonjour-S] utilisent le registre de machine pour ajouter ou supprimer des machines. Ce découplage basé sur le patron de « tableau blanc » (*White Board Pattern*) offre la possibilité de spécialiser la découverte en fonction du contexte d'exécution.

L'exemple de la Figure 56 montre que la machine C est connectée à toutes les autres machines grâce aux trois protocoles nommés précédemment. Cependant cette interconnexion n'est possible que si la topologie du réseau le permet, et dans ce cas précis, si la topologie est l'une des quatre présentées dans la Figure 55. Il est intéressant de constater que l'utilisation de protocole de découverte permet de définir un sous-graphe d'un réseau

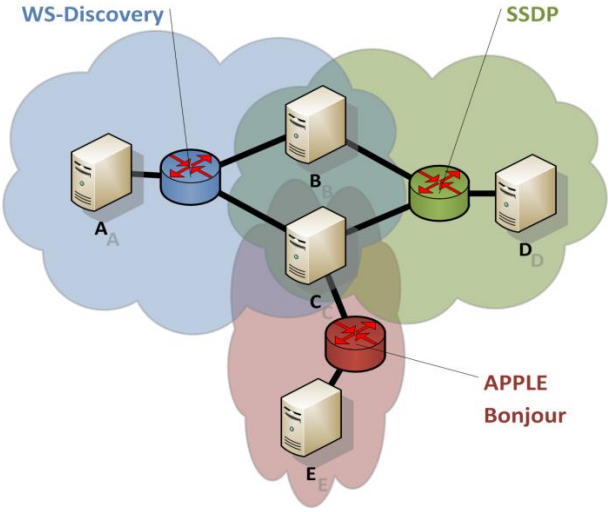


Figure 56 Recouvrement de protocoles de découverte

Comme montre la Figure 56, une même machine peut être découverte par plusieurs protocoles. Ce recouvrement de protocoles peut entrainer une concurrence entre ceux-ci. Chaque machine distante est enregistrée une et unique fois dans le registre de machine, la distinction entre les machines se faisant par leur identificateur. Le fait qu'un protocole de découverte découvre et enregistre une machine déjà présente dans le registre, n'a pas d'impact et ne pose donc pas de problème. Par contre le désenregistrement lui est problématique. Plusieurs cas se présentent :

- La machine B est éteinte, alors la machine C sera notifiée de son départ par le protocole WS-Discovery et par celui de SSDP, et devra donc la supprimer du registre de machine C.
- Les machines B et C appartiennent toutes deux à deux réseaux distincts « 10.0.0.0 » et « 192.168.0.0 » (cf. schéma 2,3 ou 4 de la Figure 55) : si le réseau « 10.0.0.0 » venait à tomber (panne du routeur par exemple), alors les machines appartenant à ce réseau seront notifiées de cette panne grâce au mécanisme de « battement de cœur » (*heartbeat*) du protocole SSDP. Les machines B et C sont donc supprimées du registre de la machine D car elles ne sont plus

accessibles ; cependant les machines B et C ne doivent supprimer que la machine D car elles restent connectées entre elles via le réseau « 192.168.0.0 ».

- Le protocole WS-Discovery est désactivé de la machine B, ce qui entraine une notification de départ aux autres machines bien que la machine B reste active. La machine A ne fait pas de supposition et supprime la machine B de son registre. Par contre la machine C continue d'être notifiée de la présence de B via le protocole SSDP.

Les exemples ci-dessus montrent deux choses :

- une machine peut être connectée à un ou plusieurs réseaux et donc à une autre machine via plusieurs « chemins » ;
- en fonction des scénarii, les notifications de départ ne sont pas nécessairement cohérentes avec la réalité de l'infrastructure.

C'est pour cela que dans l'AM **la réification et la suppression d'une machine** dans le registre ne **sont** pas **effectuées** directement par les protocoles mais **par le registre lui-même**.

Chaque protocole de découverte notifie les arrivées et départs au registre qui se charge alors des enregistrements et des suppressions nécessaires.

Notifications

Dans l'approche SAM, le contexte d'exécution des AM est évolutif : une AM peut apparaitre ou disparaitre de façon intempestive. Par conséquent le registre de machine de chaque AM émet des notifications d'arrivée, de départ et de modification de propriétés.

Il est possible de souscrire aux notifications d'une machine distante.

Réutilisation du protocole de communication

L'accès aux machines distantes par les extensions est pris en charge par l'AM. Par conséquent une AM fournit les proxies vers les machines distantes.

Nous verrons dans la section 6 de ce chapitre, que le protocole de communication entre machines peut être réutilisé pour la distribution des extensions.

4.2.2 Plate-forme à la carte

Comme le montre la Figure 54, l'AM n'est pas nativement une plate-forme d'exécution. Elle fournit les fonctionnalités nécessaires à la distribution d'un environnement. Nous pourrions très bien imaginer qu'un environnement de développement puisse être au-dessus d'une AM.

Pour cela l'AM peut être spécialisée par des extensions. Une AM ne peut avoir plusieurs fois la même extension. L'unicité d'une extension sur une AM est garantie par son identificateur ; en effet chaque extension possède un identificateur propre. L'accès aux extensions se fait via le registre d'extension. Une extension est définie par un identificateur, un nom et un ensemble de propriétés.

L'*Abstract Machine* a pour but de fournir une plate-forme distribuée extensible dynamiquement. C'est-à-dire qu'il est possible d'ajouter ou de supprimer à l'exécution de la plate-forme zéro ou plusieurs extensions. L'idée sous-jacente est que l'AM soit une plate-forme spécialisée en fonction du temps pour être au plus près des besoins.

L'AM notifie l'arrivée et le départ des extensions ainsi que la modification des propriétés d'une extension.

4.2.3 Spécialisation SAM

SAM-RT est une spécialisation de l'*Abstract Machine*, qui permet l'intégration de SOA existants dans un environnement d'exécution homogène. Nous verrons dans la section 5 de ce chapitre que *SAM-RT* permet d'intégrer des SOA centralisés comme des SOA distribués.

L'intégration de SAM-RT dans l'AM est toute simple : chaque registre (*Specification*, *Implementation* et *Instance*) est considéré comme une extension : il a un identificateur, un nom et le mécanisme d'évènement de la Figure 50 est celui fourni par l'AM.

Cependant, l'intégration va au-delà car ces trois registres sont accessibles à distance. Il est donc possible d'accéder à la liste des services présents sur une SAM-RT à distance. Cette distribution des registres de SAM-RT soulève une question qui est celle de la distribution des services. Prenons l'exemple de la Figure 57.

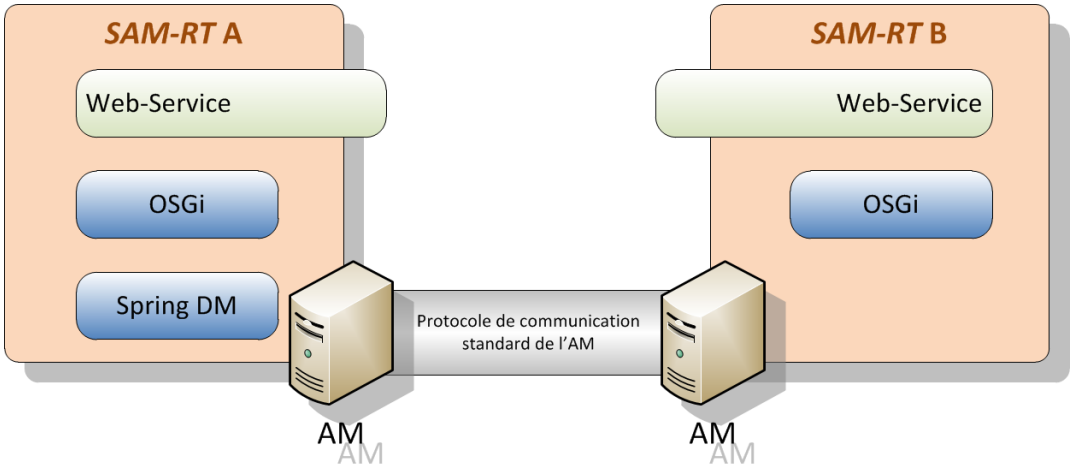


Figure 57 Exemple de deux *SAM-RT* interconnectées via leur AM

La *SAM-RT* B intègre deux technologies : OSGi, et une technologie composant à Web-Service. La *SAM-RT* A intègre trois technologies : OSGi, Spring DM et une autre technologie composant à Web-Service.

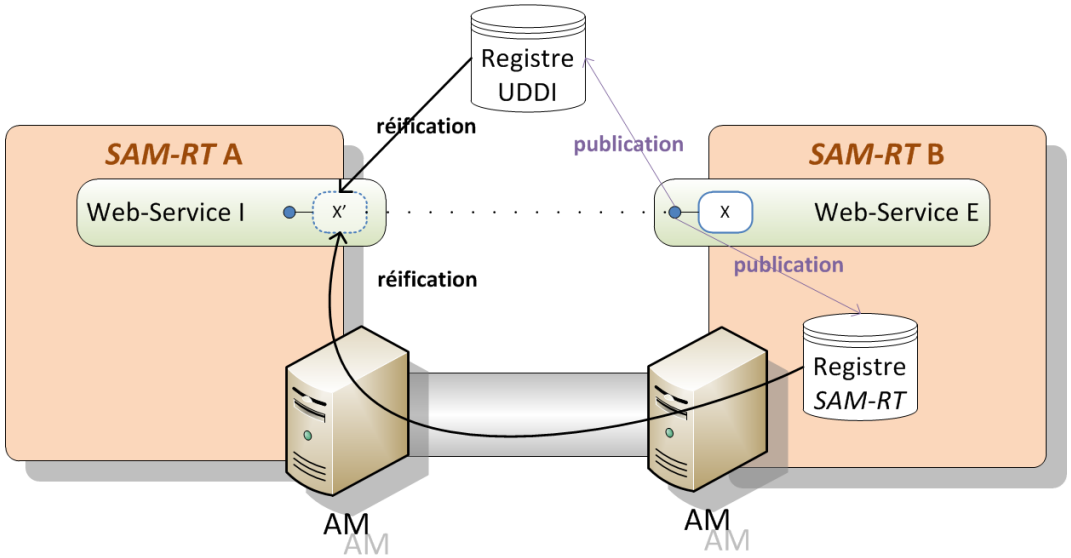


Figure 58 Identification d'un même service par plusieurs chemins

Prenons le cas de la Figure 58 : l'environnement Web-Service de SAM-RT B publie ses Web-Services dans un registre UDDI alors que l'environnement Web-Service de SAM-RT A importe (réifie) les services du registre UDDI. Par conséquent, le composant à service X de B est réifié dans A.

Théoriquement un Web-Service importé a une disponibilité dynamique ou semi-dynamique car l'environnement n'a pas accès à la gestion du cycle de vie du service ; cependant dans ce cas, les SAM-RT interconnectées via leur AM forment un environnement d'exécution homogène. Dans le cas de la technologie des Web-Services, un service peut être identifié à l'aide de son *Endpoint* (point d'invocation). SAM-RT A peut donc demander aux AM connectés si un AM est la source de ce service ; SAM-RT B lui répond alors ce qui permet à SAM-RT A d'accéder au service X (sous ses trois matérialisations) et donc aux opérations de manipulation du service.

Les trois matérialisations d'un service sont accessibles à distance, cependant les opérations de manipulation ne sont pas forcément toutes autorisées, soit pour des raisons de droit soit pour des raisons de sérialisation des paramètres. Dans cette thèse ainsi que dans l'implémentation pour la validation, l'aspect gestion des droits utilisateurs n'est pas traité ; nous limitons donc l'accès aux opérations en fonction de la sérialisation. L'AM fournit une fonctionnalité qui permet de définir si un service peut être exporté à l'aide du protocole de communication natif de l'AM. Par conséquent, si le composant à service X est exportable alors un client sur SAM-RT A peut y accéder par plusieurs chemins : par le protocole Web-Service ou par le protocole de communication de l'AM. Le fait d'avoir un environnement distribué homogène permet d'identifier le service X importé via les Web-Services comme identique à celui importé par l'AM, il n'y a donc pas duplication du proxy, ni génération de boucle.

La question de savoir quel protocole de communication doit être choisi dans le cas où il existe plusieurs chemins vers un service, ne sera pas traitée dans cette thèse.

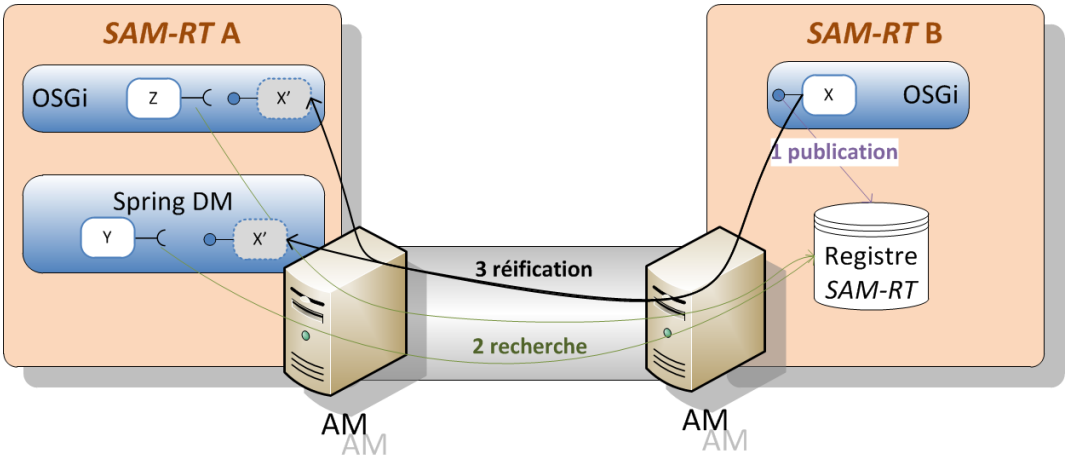


Figure 59 Distribution de services de SOA centralisés

Nous avons vu dans la Figure 58, le cas des services nativement distribués. Dans la Figure 59, nous voyons le cas d'un service d'une technologie centralisée – dans le cas ici présent : la technologie OSGi – qui est exporté via le protocole de l'AM. Dans le cas d'une technologie centralisée tel qu'OSGi, iPOJO ou Spring DM le choix des identifiants est limité car ils sont restreints par la portée du service. Dans le cas d'OSGi, l'identifiant unique est le « *service.id* », cet identifiant est attribué au moment de l'enregistrement du service dans le registre, c'est tout simplement un compteur. Ceci a deux conséquences : entre deux exécutions un même service n'aura pas nécessairement le même identifiant ; deuxièmement – et c'est le cas qui nous intéresse – il est plus que probable que deux services distincts aient le même identifiant dans deux OSGi distincts. Le fait d'avoir une AM permet de pallier à ce problème : l'identifiant du service est unique dans SAM-RT, l'identifiant de l'AM est unique sur le réseau ; par conséquent, le service identifié à la fois par l'identifiant de l'AM et l'identifiant du service

dans cette AM est unique sur le réseau. UPnP et DPWS utilisent le même principe : un service a un identifiant unique sur l'équipement, l'équipement a un identifiant unique sur le réseau donc on peut identifier le service sur le réseau.

Itinérance (*Roaming*)

Le problème de l'itinérance (*roaming*) dans SAM-RT reste un problème complexe. En effet dans le cas de technologies nativement distribuées, l'identifiant des services est défini de façon à ce qu'il soit unique sur le réseau.

Un service nativement distribué (un WS ou un appareil mobile par exemple) a un identifiant unique sur le réseau. Par conséquent si un tel service est accessible dans SAM-RT A puis disparait de A pour réapparaitre dans B, on est en droit de penser que c'est le même service Cependant l'identifiant SAM du service lui a changé car il est basé sur l'identifiant de l'AM. Inversement, dans le cas de technologies centralisées les identifiants sont uniques à la plate-forme seulement. Par conséquent si un service disparait d'une machine et le même identifiant apparait sur une autre, c'est très probablement une coïncidence et non une migration du service et de ses états sur une autre plate-forme. Dans tous les cas on est dans l'incapacité de déterminer à l'aide des identifiants si un service a migré ou si c'est une coïncidence.

Dans cette thèse nous limitons l'itinérance aux machines et non aux services.

4.3 SYNTHESE : ARCHITECTURE DE L'*ABSTRACT MACHINE*

L'Abstract Machine permet de définir un environnement distribué homogène et dynamique. Son mécanisme d'extension permet de la spécialiser selon les besoins.

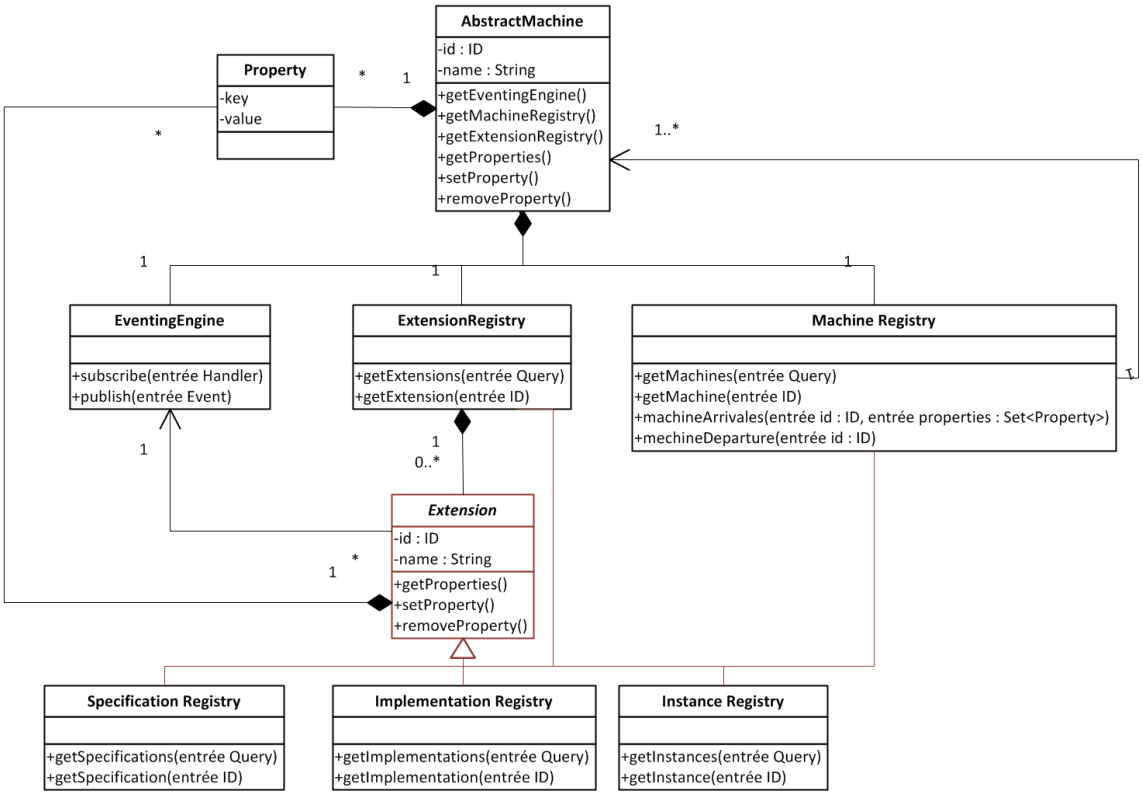


Figure 60 Méta-modèle de l'*Abstract Machine* avec les trois registres de *SAM-RT*

Dans notre cas, l'AM est étendu pour en faire un environnement d'exécution de services extensible :

- Le mécanisme de distribution permet d'identifier la source d'un service, d'éviter les duplications et les boucles. La réutilisation du protocole de communication permet dans le cas de SAM-RT de distribuer des services qui nativement ne l'étaient pas.
- Le mécanisme de notification permet de notifier la disponibilité dynamique des différents éléments de l'environnement d'exécution : machine, extension, spécification, implémentation et instance.

Nous pouvons constater qu'il y a un concept implicite de portée. Dans notre cas nous ne masquons pas la distribution, seulement la technologie ; cependant il serait intéressant de définir un mécanisme qui permette d'agréger un ensemble de machines à l'aide d'un filtre par un représentant. Par exemple nous souhaitons rechercher tous les services d'impression des AM qui possèdent l'extension service. Cependant ce sujet ne sera pas traité dans le cadre de cette thèse.

5. ABSTRACTION DES TECHNOLOGIES : INTEGRATION VERTICALE

L'environnement SAM a pour but de définir un environnement d'exécution unifiant les différentes technologies SOA dans un modèle homogène ; d'où son nom *Service Abstract Machine*. Cependant la vision sous-jacente est celle de l'« approche globale » ; par conséquent, « SAM » désigne l'approche. Quant à lui, l'environnement d'exécution est nommé *SAM-RT*.

L'approche SAM définit un méta-modèle cœur : *SAM CORE*, qui peut être étendu en fonction des différents aspects non-fonctionnels et / ou de la spécialisation pour un modèle métier donné. Comme il a été vu dans la section 1.3 de ce chapitre, l'environnement d'exécution est composé d'un ensemble de sous environnements prenant en charge les différents aspects. *SAM-RT* désigne l'environnement d'exécution ; c'est-à-dire l'ensemble de tous les environnements dédiés à un aspect du contexte d'exécution. Le sous-environnement dédié à l'aspect de service défini par *SAM CORE* se nomme *SAM-CORE-RT*.

La section 3 de ce chapitre met en évidence l'écart « d'interprétation » entre le concept de service au développement et le concept à l'exécution d'un même méta-modèle. Or un des buts de *SAM-CORE-RT* est de prendre en charge et de masquer les technologies SOA hétérogènes existantes. Nous nous intéresserons donc dans cette section à *SAM-CORE-RT* et à l'intégration des technologies existantes dans l'environnement d'exécution. Dans un premier temps, nous aborderons les mécanismes d'intégration et donc de l'architecture de *SAM-CORE-RT*. Puis, dans un second temps, nous nous intéresserons à l'alignement sémantique entre les concepts des technologies SOA existantes et les concepts définis par le méta-modèle de SAM-CORE (exécution).

5.1 ARCHITECTURE D'INTEGRATION

Il existe couramment une multitude de technologies à service et leur nombre ne cesse de croitre. Il est bien évident qu'on ne cherche pas à intégrer toutes les technologies existantes dans un même environnement, mais on souhaite les intégrer en fonction des besoins. En d'autre termes l'environnement d'exécution doit pouvoir être configuré en fonction des besoins de ou des applications. Par conséquent, l'architecture de *SAM-CORE-RT* doit être modulaire pour permettre la définition de configuration « à la carte ». Il faut donc identifier les différents modules rentrant en jeu. Pour cela il faut dissocier les parties qui sont spécifiques aux technologies de ce qui est générique. De même, il faut distinguer ce qui est spécifique à SAM-CORE-RT de ce qui est spécifique à la technologie.

Nous avons identifié trois niveaux correspondant à la combinaison des différentes propriétés ci-dessus : la partie technologies à service (spécifique), la partie SAM-CORE-RT spécifique à l'intégration d'une technologie et la partie SAM-CORE-RT générique aux technologies (cf. Figure 64).

5.1.1 Technologies à service

Les différentes technologies à services intégrées dans SAM-CORE-RT ne sont pas couplées à cette dernière. En effet celles-ci fonctionnent indépendamment de notre environnement. Cependant dans les technologies actuelles, il existe parfois des ponts entre elles : par exemple, il existe des ponts entre la technologie OSGi [OSGia] et la technologie UPnP [OSGib] ; de même que pour la technologie DPWS [BSSG08] [OSGic]. Que se passe-t-il si nous intégrons deux technologies à service alors qu'il existe un pont entre celles-ci ? Cela pose un ensemble de problèmes qui se révèle être toujours un problème d'identification lié à la distribution de l'espace d'exécution (cf. section 4 de ce chapitre).

Dans la suite de ce chapitre, nous considérerons que les espaces d'exécution sont distincts ; c'est-à-dire que les « images » (proxies / représentation de service distant) de service provenant d'un autre espace d'exécution ne sont pas considéré comme des services mais comme des supports de communication.

5.1.2 SAM-CORE-RT générique

SAM-RT est un SOA abstrait dont l'espace d'exécution est l'union des espaces d'exécution des technologies intégrées. Les représentations des services sont remontées dans SAM-CORE-RT alors que les opérations de navigation et de manipulation sont déléguées aux SCM concernées par un module d'*intégration*. Par conséquent, SAM-CORE-RT générique (cf. Figure 61) contient :

- le méta-modèle SAM-CORE ;
- l'ensemble des mécanismes propres au SOA SAM, c'est-à-dire :
  - les trois registres : spécification, implémentation et instance,
  - le mécanisme de notification,
- le mécanisme d'intégration / de délégation entre le niveau générique et le niveau spécifique (SCM).

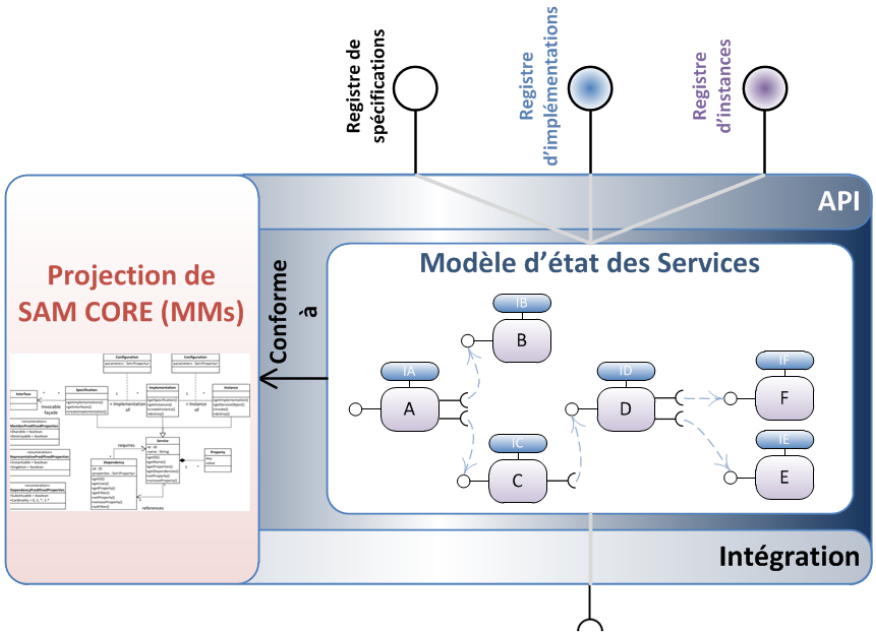


Figure 61 Partie générique de SAM-CORE-RT

5.1.3 SAM-CORE-RT spécifique

Les technologies à Service existantes sont intégrées dans SAM-CORE-RT par des modules du nom de *Service Concrete Machine* (SCM), nommé ainsi en contraste à la *Service Abstract Machine* (SAM). Ces modules ont pour but d'aligner la sémantique entre le méta-modèle *SAM-CORE* et le méta-modèle de la technologie à Service cible. Plus simplement, il fait le pont entre l'espace d'exécution d'une technologie et celui de la SAM (cf. section 5.1.2).

Il y a un alignement sémantique à la fois du méta-modèle SAM-CORE vers le méta-modèle de la technologie cible, et réciproquement de la technologie cible vers SAM-CORE. En effet le modèle descriptif obtenu par introspection de la technologie cible peut être manipulé. Nous avons vu dans la section 2.3 de ce chapitre qu'il était possible de réaliser un certain nombre d'opérations de manipulation liées au cycle de vie d'un composant orienté service. De ce point de vue le modèle obtenu par introspection est aussi prescriptif car il permet d'agir sur l'espace d'exécution de la SAM.

Par conséquent, une SCM doit garantir la « validité » (cf. Figure 18) et la « correction » (cf. Figure 19) entre l'état d'exécution du système cible et le modèle homogène que l'on nomme modèle d'état.

Matérialisation : intégration de la plate-forme à service cible

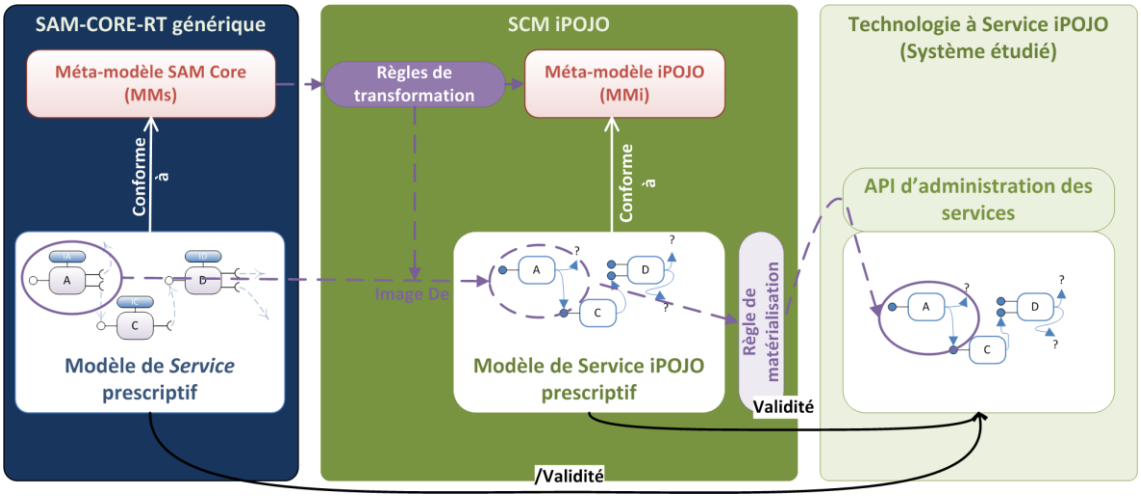


Figure 62 Validité : SCM

Les opérations de manipulation sont des opérations de modification de l'architecture de l'état d'exécution. Modifier l'architecture nécessite d'accéder aux opérations d'administration du cycle de vie des services ; or ces opérations sont généralement fournies non pas par les services mais par les plates-formes. Par conséquent SAM-RT intègre les services **et** leurs plates-formes.

La matérialisation se fait par l'intégration de la plate-forme de la technologie cible afin d'accéder aux opérations de gestion du cycle de vie d'un service.

Dans notre cas le méta-modèle utilisé n'est pas celui du système étudié. Pour pouvoir matérialiser des éléments dans l'espace d'exécution d'un système, il est donc nécessaire de connaitre son méta-modèle. La plupart des plates-formes ont un méta-modèle implicite – pas fourni. Par conséquent, il faut que la SCM connaisse le méta-modèle de la technologie cible. Par conséquent, la validité du SOA étudié par rapport au modèle prescriptif nécessite une transformation de modèle.

Pour pouvoir garantir la « validité », la SCM doit avoir (cf. Figure 62) :

- Le méta-modèle du concept de service de la plate-forme ciblée ;
- Accès aux opérations d'administration de la plate-forme.

Représentation : intégration des services de la plate-forme cible

La représentation se fait, quant à elle :

- par l'introspection de l'état des services de l'environnement d'exécution de la technologie cible ;
- par la caractérisation des services en fonction de la technologie cible.

Les technologies intégrées sont des SOA ; l'utilisation de leur registre permet de découvrir l'ensemble des services disponibles de leur espace d'exécution. Chaque service est alors *représenté par* trois éléments dans le modèle descriptif : la spécification, l'implémentation et l'instance définie dans SAM-CORE.

Cependant, l'assignation des valeurs des propriétés prédéfinies de SAM-CORE est plus compliquée. En effet les propriétés peuvent être propres à la technologie comme propres au service.

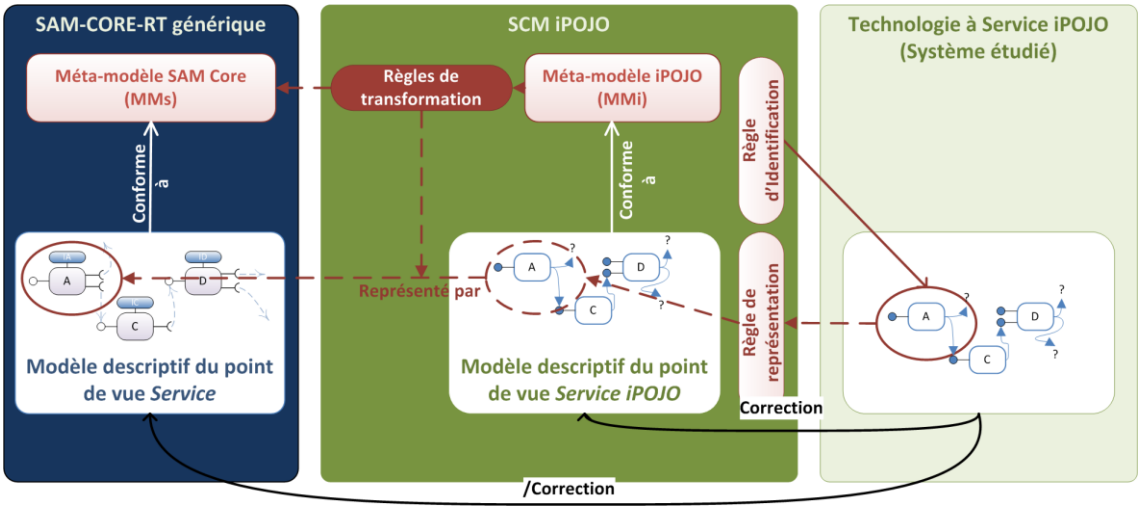


Figure 63 Correction : SCM

Prenons le cas d'une SCM de *Web Services* : les services sont intégrés en créant des proxies pour chaque service trouvé dans le registre UDDI ciblé. La représentation de l'implémentation dépend des règles de représentation définies par la SCM. Par exemple une SCM WS définira le code du proxie comme étant l'implémentation alors qu'une autre le représentera par une classe de représentation quelconque. Par conséquent, il n'y a pas de sens à instancier un nouveau représentant (proxy,…) du service car ils représentent tous au final le même *Web Service* et donc l'identificateur est le même. Dans le cas l'utilisation de *Web Service*, une SCM définira systématiquement la propriété « implementation.instantiable = false » car elle est intrinsèque à la technologie. Alors que dans le cas de la technologie iPOJO le fait qu'une implémentation de service soit instanciable, ou non, dépend de l'implémentation.

Pour pouvoir garantir la « correction » du SOA étudié (cf. Figure 63), la SCM doit pouvoir représenter chaque élément identifié (service) dans le modèle descriptif conformément au méta-modèle SAM CORE. Cependant, nous verrons dans la section 5.2 de ce chapitre, que le problème n'est pas de représenter un élément du système, mais de l'identifier. De même que pour la validité, la correction du modèle par rapport au système est basée sur une transformation de modèle

5.1.4 Intégration de la SAM dans une technologie cible

L'environnement d'exécution SAM-CORE-RT peut être architecturé en trois niveaux (cf. Figure 64):

- Le niveau exécution qui est l'union des environnements d'exécution ;
- Le niveau SAM-CORE-RT spécifique qui est l'union des SCM ;
- Le niveau SAM-CORE-RT générique qui fournit le SOA abstrait.

Le but de l'approche SAM est de pouvoir définir des applications composées de services dynamiques et hétérogènes. Nous avons vu dans les sections précédentes comment on intègre les services dans un modèle commun (SAM-CORE) mais pas comment ils interopérent. Comme expliqué dans la section 5.1.1 de ce chapitre, les espaces d'exécution des technologies à service sont disjoints, Par conséquent, pour pouvoir faire interagir deux services provenant de deux espaces distincts il est nécessaire de mettre en place des mécanismes de communication entre ceux-ci. La méthode retenue est d'instantier dans un espace d'exécution des représentations (proxies) de services venant d'autres espaces. Ce travail est effectué une nouvelle fois par les *Service Concrete Machines*.

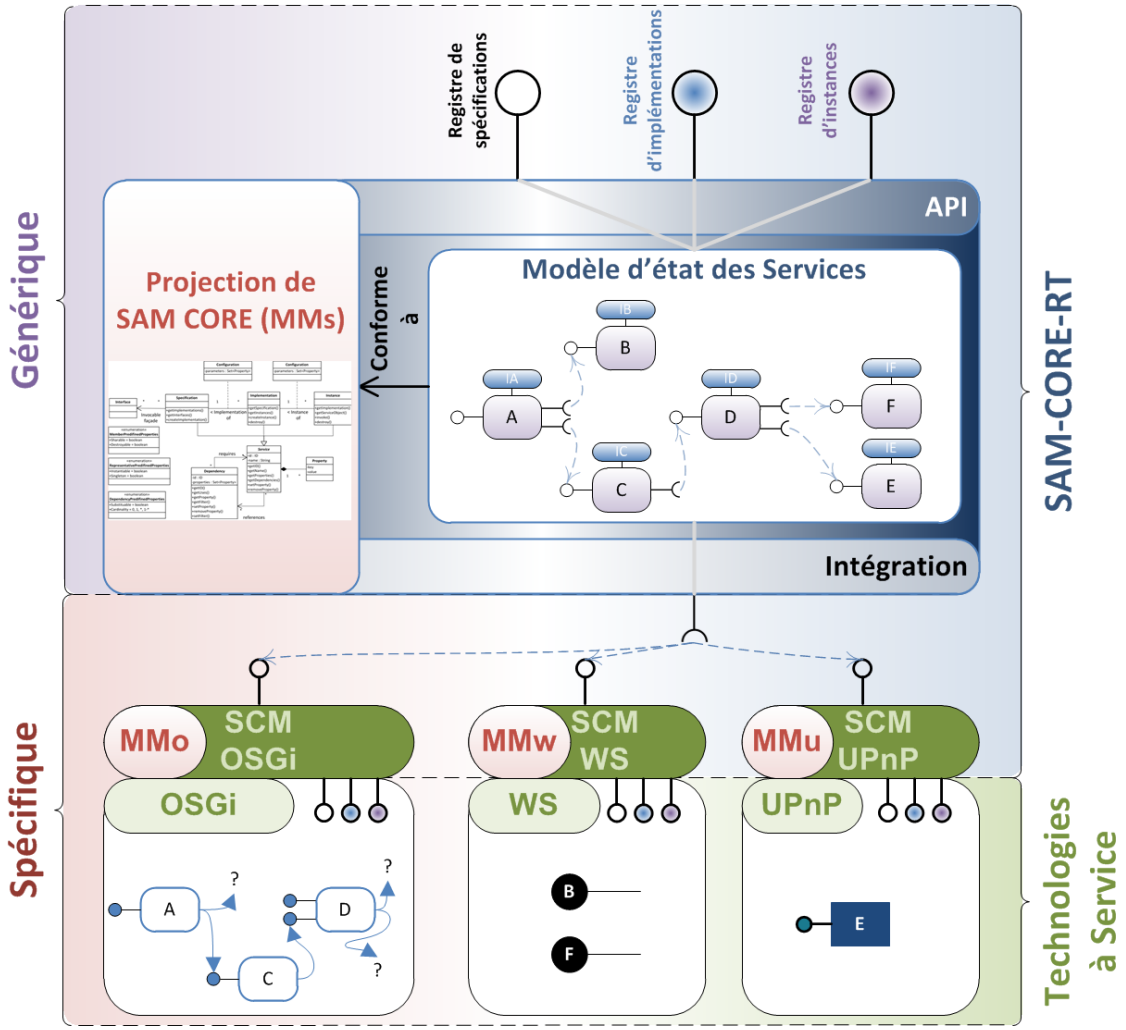


Figure 64 Architecture de SAM-CORE-RT

Il faut considérer deux cas :

- Un service est développé pour une technologie cible indépendamment de SAM , ce qu'on appelle courrament un « service *legacy* » ;
- Un service est développé pour une technologie cible intégrée dans SAM , qui sera appelé « service SAM ». Les registres de SAM sont :
  - présentés comme des services de la technologie cible ;
  - accessibles via les annotations (cf. 0 de ce chapitre) définies par l'API de SAM.

Dans le cas d'un dépendance d'un service SAM, SAM-CORE-RT se charge de fournir le proxy de communication vers le service correspondant. Ce problème est plus compliqué dans le cas d'un « service *legacy* », les technologies intégrés ne sont pas prévus nativement pour l'être et ne fournissent donc pas de mécanisme pour la création de proxy de communication vers une autre technologie. Pour cela, il faut soit modifier la plate-forme cible, soit définir un mécanisme transparent basé sur l'architecture de la plate-forme cble.

Dans le premier cas, il faut modifiée la plate-forme cible pour que les registres pertinents de SAM (généralement seulement celui d'instance) soient intégrés dans les registres correspondants de la plate-forme cible. Cette méthode permet de délégué le calcul des requêtes au module SAM-CORE-RT générique ; et par conséquent elle optimise la gestion mémoire et évite la duplication de code au détriment du processeur.

Dans le second cas, il faut définir un mécanisme transparent basé sur l'architecture de la plate-forme cible. Pour la plupart des technologies à service, on peut utiliser la façon suivante : la SCM enregistre les proxies de communication de tous les services présents dans SAM dans les registres correspondants de la plate-forme cible. Cette seconde méthode, quant à elle, facilite l'implémentation mais elle est catastrophique en termes de consommation mémoire. En effet dans le pire des cas chaque service sera représenté dans chaque espace d'exécution, que celui-ci soit utilisé ou non.

La première méthode est la plus intéressante d'un point de vue exécution ; mais elle est conditionnée au fait qu'il soit possible de modifier la technologie à service utilisé ; cela suppose qu'on ait le code source de la plate-forme et qu'il soit techniquement possible de l'implanter.

En résumé, l'intégration des SOA existants permet d'obtenir une représentation homogène de l'état d'exécution de ceux-ci, mais ne permet pas leur interopérabilité. Pour cela il faut intégrer le SOA SAM dans chaque environnement d'exécution des SOA existantes pour pourvoir les faire interopérer. Tout élément d'un espace d'exécution peut accéder à tous les éléments des espaces d'exécution intégrés par la SCM via la SCM.

L'approche d'intégration proposée par la SAM est similaire à un routeur (SAM-RT) connecté à différents réseaux (espaces d'exécution) via un support de communication (média et protocole) (SCM), tel que schématisé dans la Figure 65.

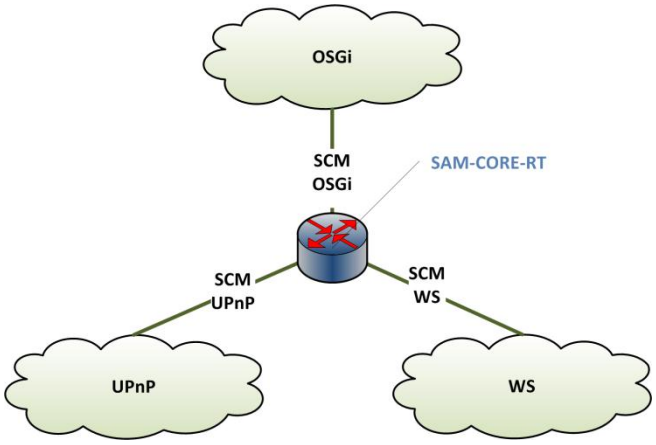


Figure 65 Principe d'interopérabilité de SAM-RT schématisé selon une topologie réseau

5.2 ALIGNEMENT SEMANTIQUE

SAM CORE est un méta-modèle unificateur de l'ensemble des technologies SOA car celle couvre l'ensemble des concepts liés aux services définis par les SOA existants. Cependant, un bon nombre de technologies ne fournissent pas tous les concepts et les informations de *SAM CORE*; et d'autre part, elles ne fournissent pas nécessairement les opérations de manipulation du cycle de vie de leurs services ; soit parce qu'elles ne sont pas accessibles, soit parce qu'elles ne sont pas autorisées. Ces écarts aussi bien du point de vue conceptuel qu'opérationnel posent un ensemble de problèmes liés à l'alignement sémantique.

Dans un premier temps nous allons étudier l'impact de la distribution des espaces d'exécutions des technologies « concrètes » à service. Puis l'alignement sémantique des concepts de technologies qui nous semblent représentatives.

5.2.1 Distribution des espaces d'exécution

La Figure 66 exhibe les espaces d'exécution de la Figure 54. Nous appelons *SAM-RT** le sous-ensemble de AM* ayant au moins l'extension *SAM-CORE-RT*. En d'autres termes, SAM-RT* désigne les graphes dont les nœuds contiennent *SAM-CORE-RT* (pour plus de détails voir l'Annexe A -).

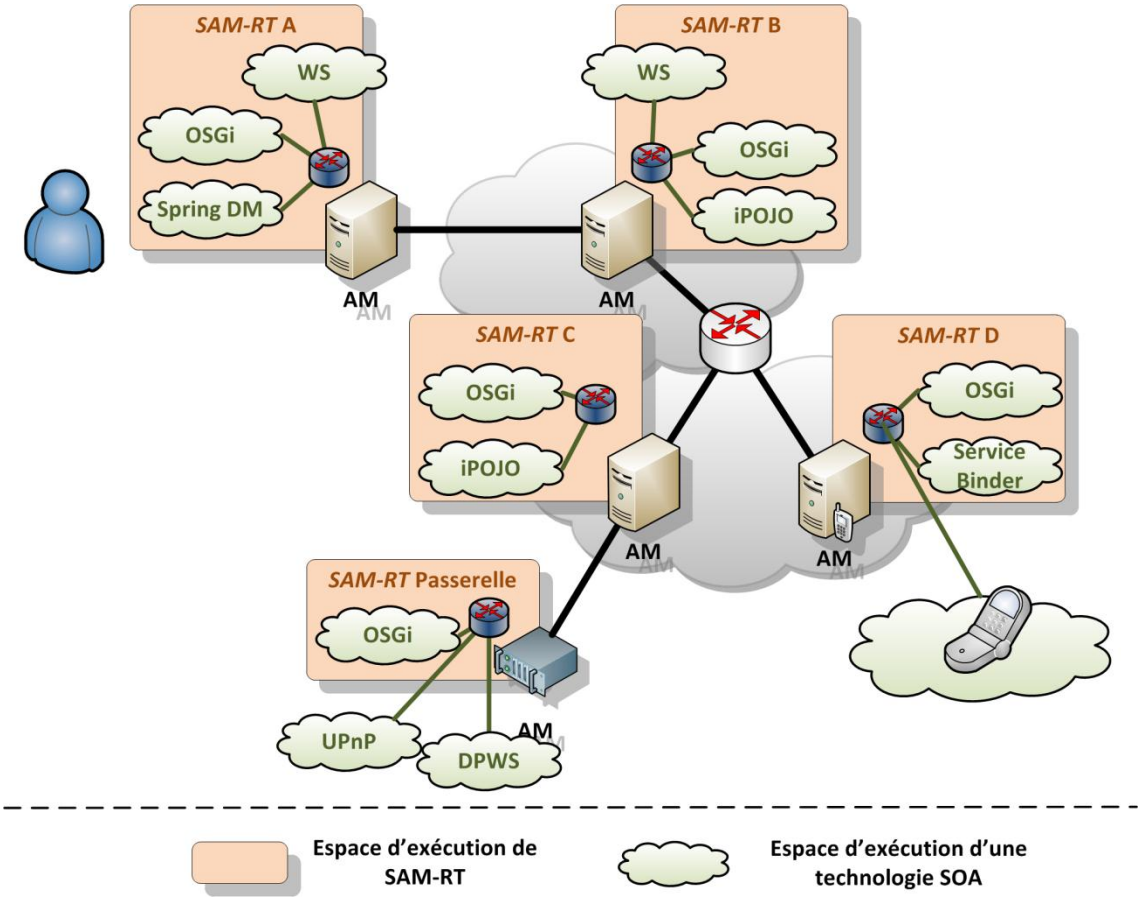


Figure 66 Exemple de SAM-RT*

La Figure 66 met en évidence plusieurs cas :

1. L'espace d'exécution d'une technologie « concrète » à service est pleinement inclus dans l'espace d'exécution de SAM-RT ; c'est le cas des technologies OSGi, iPOJO, Spring DM et Service Binder.
2. L'espace d'exécution d'une technologie « concrète » à service est hors de l'espace d'exécution de SAM-RT ; c'est le cas des technologies UPnP et DPWS. Les services de ces espaces sont alors importés / réifiés dans la SAM-RT.
3. L'espace d'exécution d'une technologie « concrète » à service est partiellement inclus dans l'espace d'exécution de SAM-RT* ; c'est le cas de la technologie Web-Service. C'est-à-dire que la source du service peut être sur la SAM-RT locale, sur une SAM-RT distante, ou que la source n'est pas dans SAM-RT* et est donc importée/réifiée.

Dans le cas 1, la plate-forme et ses services sont intégrés dans l'environnement d'exécution de SAM-RT et il est possible d'accéder aux opérations d'administration du cycle de vie d'un service. Les opérations de manipulation sont donc théoriquement autorisées (on peut installer, instancier et détruire un service iPOJO par exemple).

Dans le cas 2, l'environnement d'exécution de la technologie concrète n'appartient pas à SAM-RT* et par conséquent les opérations d'administrations du cycle de vie ne sont pas accessibles (on ne peut pas créer / détruire un capteur UPnP par exemple).

Le cas 3 est la combinaison des deux précédents cas. C'est-à-dire qu'un ensemble de services appartient à l'espace d'exécution de SAM-RT* et donc les opérations d'administrations de leur cycle de vie sont théoriquement accessible ; et que l'espace d'exécution est partiellement hors de l'espace d'exécution de SAM-RT*.

Ces cas mettent en évidence le lien entre la gestion du cycle de vie d'un service et sa disponibilité dynamique.

Gestion du cycle de vie d'un service et disponibilité dynamique

Un service n'appartenant pas à l'espace d'exécution de SAM-RT*, ne peut pas être administré (au sens cycle de vie). Les opérations de manipulation sur l'architecture (*createInstance, dispose, etc*) ne sont donc pas autorisées. Son cycle de vie ne pouvant être contraint, ce service a donc théoriquement une disponibilité dynamique ou semi-dynamique.

Inversement, un service appartenant à l'espace d'exécution de SAM-RT* peut en théorie être administré par conséquent sa disponibilité est statique ou semi-statique. Les opérations d'administration sont théoriquement autorisées mais sont conditionnées à leur alignement entre la technologie concrète et *SAM-RT*. Par exemple, l'instanciation d'un service dans OSGi n'est pas spécifiée dans le CORE. La politique d'instanciation et la gestion du cycle de vie étant à la discrétion du développeur du service il n'est pas possible d'instancier un service OSGi à partir de son implémentation (ou du moins de façon très hasardeuse) ; en effet la gestion du cycle de vie du service est défini directement par le code en fonction du cycle de vie du *bundle*. Cependant, *Declarative Service* définit des opérations d'administrations du cycle d vie d'un service ; pour cela le service OSGi doit implémenter l'interface *ComponentFactory*[14] (permet l'instanciation par une tierce partie) voire en passant par l'interface *ServiceFactory*[15](permet de définir la politique d'instanciation sessionelle). Cet aspect sera discuté dans la section 5.2.3 de ce chapitre.

5.2.2 Alignement des méta-modèles de service

Dans cette section, nous allons distinguer les services appartenant à l'espace d'exécution de *SAM-RT** de ceux qui ne le sont pas. En effet, dans le cas d'un service appartenant *à SAM-RT**, les notions d'instance et d'implémentation ont un sens car il nous est possible de les identifier via leur plate-forme respective, ce qui n'est pas le cas des services qui sont hors de *SAM-RT**.

La Table 9 montre l'alignement sémantique de trois technologies qui appartiennent à l'espace d'exécution d'une SAM-RT : OSGi, iPOJO. Il faut noter qu'OSGi est une spécification et non une implémentation ; cet alignement est fait en fonction de leur spécification.

Première observation : le concept de spécification tel que nous le définissons n'existe pas dans ces deux technologies ; le service étant dans ce cas-là une interface Java et non l'ensemble des fonctionnalités fournies par cette implémentation. De notre point de vue, si l'implémentation fournie plusieurs interfaces c'est qu'il y a une cohérence entre-elle pour fournir un service plus général. S'il n'y a pas de cohérence entre les fonctionnalités alors il y a un problème dans la modularisation et l'implémentation est donc à revoir. En d'autres termes dans OSGi et iPOJO une implémentation peut fournir plusieurs services définis par une interface Java ; dans le cas de SAM, nous spécifions un service

[14] http://www.osgi.org/javadoc/r4v42/org/osgi/service/component/ComponentFactory.html
[15] http://www.osgi.org/javadoc/r4v42/org/osgi/framework/ServiceFactory.html

ayant des fonctionnalités décrites par zéro ou plusieurs interfaces Java et donc une implémentation implémente un seul service.

Cependant dans le cas d'iPOJO, il est possible de définir des propriétés sur l'ensemble des interfaces Java d'un composant ; propriétés qui seront hérités par les instances des services. Nous voyons dans ces trois technologies que le concept de spécification est inexistant à l'exécution ne s'intéressant qu'aux instances, parfois aux implémentations quand elles définissent une fabrique. Cependant, dans l'approche SAM, une spécification est une entité à part entière ; SAM-RT fournit les spécifications d'un service (ensemble d'interfaces Java et propriétés) indépendamment du fait qu'il y est ou non des implémentations ou des instances.

Dans le cas des plates-formes d'exécutions OSGi, la spécification d'un service est déduite du registre, c'est l'ensemble des interfaces Java fournies par le service, les propriétés du service dans le registre sont propes quant à elles à l'instance. Par conséquent, dans le cas d'OSGi le cycle de vie d'une spécification de service dépend du cycle de vie des instances. Même si les interfaces Java sont encore disponibles, la spécification disparait si les instances qui référencent ces interfaces ne sont plus dans le registre. Dans le cas d'iPOJO cela dépend de l'implémentation de la *SCM* car il est possible d'extraire les spécifications des méta-informations contenues dans l'unité de déploiment du service, son cycle de vie peut donc être dissocié de celui de l'instance ou de la fabrique.

Dans le cas de SAM un service est un élément logiciel ayant une ou des fonctionnalités mais ne les exposants pas nécessairement ; c'est-dire qu'un client d'un service a une fonctionnalité dans une application et est donc considéré aussi comme un service. Dans le cas d'OSGi, il n'est pas possible de découvrir ni la spécification, ni l'implémentation ni l'instance d'un client de service. Dans le cas d'iPOJO la spécification et l'implémentation peuvent être déduite des méta-informations contenues dans l'unité de déploiment du client ; pour le cas de l'instance cela est plus compliqué.

Table 9 Alignement sémantique pour des services "legacy"

	OSGi [OSGia]	iPOJO
Spécification	L'ensemble des interfaces java enregistré avec le *ServiceObject*	L'ensemble des interfaces java définit dans le handler *provide*
• Instanciable	*False*	*False*
• Singleton	*True*	*True*
• Sharable	*True*	*True*
• Availability	*Dynamic,* dépend des éléments enregistrés dans le registre	*Static* ou *Dynamic* dépend de sa résolution par la SCM
Implémentation	la classe du *ServiceObject*	*Component*
• Instanciable	*False*	Dépend si le composant est public ou privé
• Singleton	*False*	*False*
• Sharable	*False*	Dépend si le composant est public ou privé
• Disposable	*False*	*False*
• Availability	*Static*	*Static*
Instance	Le *ServiceObject*	*Instance*
• Disposable	*False*	Seulement par celui qui l'a instancié
• Sharable	*True*	Dépend de la politique d'instanciation
• Availability	*Static*	*Static*
Dépendance	En pratique dans OSGi les	Le Handler *Require*

	dépendances sont implicites	
• cardinality	*Undefined*	0-1, 1, * sont autorisés et dépendent de l'implémentation du service requéreur
• addable	*Undefined*	*True* dans le cas d'une dépendance multiple
• removable	*Undefined*	*True* dans le cas d'une dépendance multiple
• modifiableFilter	*Undefined*	false

La valeur *Undefined* de la Table 9 indique que les propriétés sont possible mais que l'information n'est pas fournie par la plate-forme.

Les services dont l'espace d'exécution n'est pas hébergé par une SAM-RT peuvent être intégrés par des SCM. Ces services sont alors importés / réifié dans un espace d'exécution hébergé par SAM-RT. Cette importation dépend de la *Service Concrete Machine* qui prend en charge cette technologie.

Table 10 Alignement sémantique pour des services importés

	UPnP	DPWS	WS
Spécification	Interface Java généré à partir du SCDP, ou proxy spécifique.	Interface Java généré à partir du WSDL ou proxy spécifique.	Interface Java généré à partir du WSDL ou proxy spécifique.
• Instanciable	*False*	*False*	*False*
• Singleton	*True*	*True*	*True*
• Sharable	*True*	*True*	*True*
• Availability	*Dynamic*	*Dynamic*	*Dynamic*
Implémentation • Instanciable • Singleton • Sharable • Disposable	Dépend des choix faits par la *Service Concrete Machine* en fonction des contraintes de la plate-forme		
• Availability	*Dynamic*	*Dynamic*	*Dynamic*
Instance • Sharable • Disposable	Dépend des choix faits par la *Service Concrete Machine* en fonction des contraintes de la plate-forme		
• Availability	*Dynamic*	*Dynamic*	*Dynamic*
Dépendance	Un équipement UPnP ne possède pas de dépendance	Un équipement DPWS ne possède pas de dépendance	*Undefined*

Dans le cas des équipements DPWS et des Web-Services, les services sont exposés sous la forme d'un WSDL[16]. Pour les intégrer dans SAM, il faut les transformer dans un monde Java, c'est-à-dire les

[16] *Web Service Description Language* :
version 1 http://www.w3.org/TR/wsdl

transformer en une ou plusieurs interfaces Java ; de manière similaire aux *Base Drivers* défini par OSGi [OSGib] Prenons par exemple le Base Driver DPWS [BSSG08] [OSGic] ou le Base Driver UPnP [OSGib] pour OSGi : chaque fois qu'ils trouvent un service ils génèrent un proxy générique, c'est-à-dire un proxy dont l'interface est définie statiquement et où tout l'alignement sémantique est à la charge du client ; par exemple dans le cas du DPWS du projet Amigo[17], un client recherche un service DPWS via le registre de service OSGi puis lui demande ses différents *Endpoints*. Un *Endpoint* représente le point d'invocation du service (cf. Figure 67).

```
package org.osgi.service.ws.addressing;
import org.osgi.service.dpws.DPWSService;
import org.osgi.service.ws.exception.WSException;

public interface Endpoint {

        public EndpointReference getEndpointReference();
        public MessageContent invoke(MessageContent content) throws WSException;
        public void invokeOneWay(MessageContent content) throws WSException;

}
```

Figure 67 org.osgi.service.ws.addressing.Endpoint défini et utilisé dans [BSSG08]

Cependant dans le cas des WS et donc de DPWS les paramètres d'invocation peuvent être des types complexes qui sont décrit dans le WSDL. L'invocation se fait donc sous forme d'un flux XML conforme au WSDL.

```
package org.osgi.service.ws.addressing;
import javax.xml.stream.XMLStreamReader;

public class MessageContent {

        private String m_actionURI;
        private XMLStreamReader[] m_body;


        public MessageContent(String actionURI, XMLStreamReader[] body) {
                this.m_actionURI = actionURI;
                this.m_body = body;
        }

        public String getAction() { return m_actionURI; }

        public void setAction(String actionURI) { this.m_actionURI = actionURI; }

        public XMLStreamReader[] getBody() { return m_body; }

        public void setBody(XMLStreamReader[] body) { this.m_body = body; }
}
```

Figure 68 org.osgi.service.ws.addressing.MessageContent défini et utilisé dans [BSSG08]

Par conséquent pour utiliser un tel service, il est nécessaire de connaitre à l'avance ses paramètres. Générer à l'exécution les classes correspondant au type complexe n'a pas de sens, car si celles-ci ne sont pas connues au développement et le client ne sera pas comment les utiliser.

Dans un cas plus général, réifier des services qui ne sont pas issus du monde Java nécessite de développer des proxies spécifiques qui seront utilisé à l'exécution. **Dans le cas de services importés ; c'est avant tout la SCM qui définit leurs caractéristiques** (cf. Table 10).

version 2 http://www.w3.org/TR/wsdl20/

[17] https://gforge.inria.fr/frs/?group_id=160&release_id=1804

5.2.3 Alignement des opérations d'administration

Toutes les plates-formes à service ne fournissent pas les mêmes opèrations d'administration du cycle de vie d'un service ; en effet ces opérations dépendent de différents aspects pris en compte ou non par une plate-forme. Pour expliquer cela nous allons nous intéresser au cycle de vie d'un service OSGi et celui d'un service iPOJO.

La spécification OSGi définit une plate-forme à Service permettant le déploiement dynamique de nouveau service. Par conséquent il prend en compte les aspects du déploiement tel quer l'unité de déploiement, la résolution des resources, l'activation, etc.

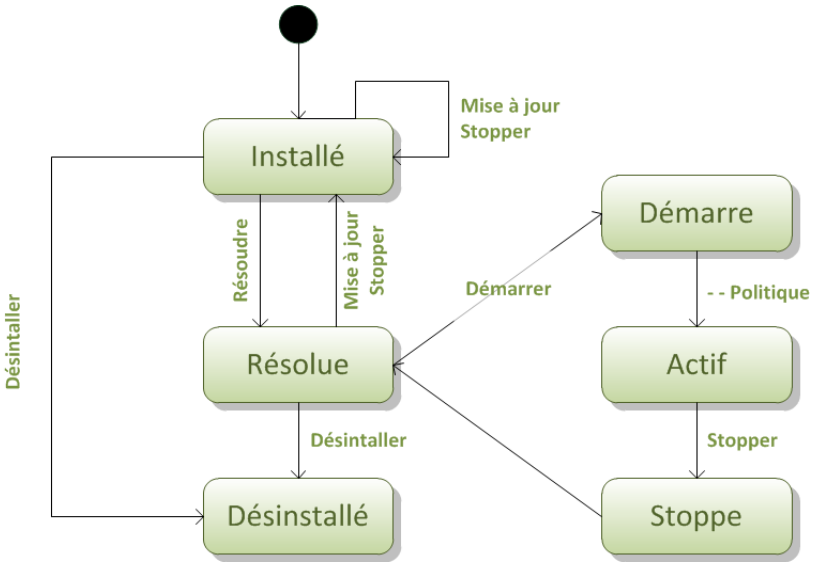


Figure 69 Diagramme de transition d'état d'un bundle définit dans [OSGia]

Par défaut – c'est-à-dire en pratique – OSGi gère avant tout des unités de déploiment nommé *bundle*. La Figure 69 définit les différents états que peut avoir un *bundle* :

- Installé : le bundle a été installé sur la plate-forme avec succès.
- Résolue : Tous les classes Java que le *bundle* requiert sont disponibles et il peut donc être démarré.
- Démarre : le *bundle* est démarré, la méthode *start* de l'activateur du bundle est appelée en fonction des propriétés définies dans la politique (voir page 100 de [OSGia]).
- Actif : l'activation du *bundle* a été réalisée avec succès ; le bundle est donc en cours d'exécution.
- Stoppe : le *bundle* est en train d'être stoppé.
- Désinstallé : le *bundle* a été désintallé et n'est donc plus disponible sur la plate-forme.

Dans la spécification OSGi, le cycle de vie des services n'est pas défini explicitement car il dépend de l'implémentation contenue dans le bundle. Les dépendances d'un bundle sont avant tout des dépendances de classe Java, alors q'une dépendance de service est une déclaration optionnelle qui n'est pas prise en charge. Nativement – spécifié dans [OSGia] – OSGi ne permet pas de gérer le cycle de vie d'un service seulement de son unité de déploiement (bundle). Par conséquent les opération de manipulation telles que createInstance ou dispose ne sont pas disponibles. Cependant OSGi définit dans [OSGib] un ensemble de spécification de services optionnels ; certaines de ces spécifications fournissent un modèle et un environnement au-dessus d'OSGi permettant de développer et d'exécuter des services avec un cycle de vie défini ; par exemple Declarative Service (dont l'origine est Service Binder [CerT04]).

iPOJO fournit aussi un modèle et un environnement au-dessus d'OSGi permettant de développer et d'exécuter des services. La Figure 70 définit les différents états que peut avoir une instance de

composant dans iPOJO. Bien que non schématisé dans [EscT08], il existe aussi un cycle de vie des composants iPOJO.

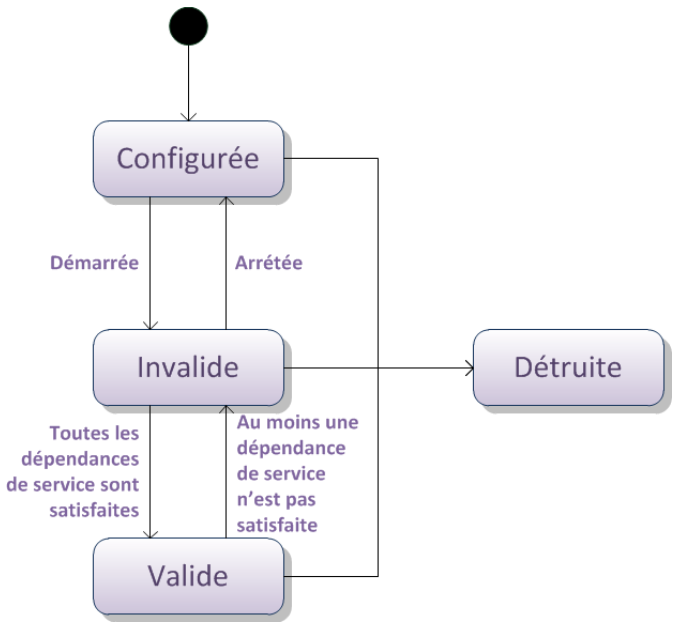


Figure 70 Cycle de vie des Instances de composant (figure 57 de [EscT08])

La différence entre OSGi Core [OSGia] et les technologies comme iPOJO, *service binder* ou *declarative service* [OSGib] se situe sur le concept d'implémentation de service. Dans iPOJO comme dans *service binder* ou *declarative service*, le concept d'implémentation est spécifié. Puisque le concept est spécifié, il est possible de l'identifier dans le système et donc pouvoir agir dessus. En effet en théorie comme en pratique, ces plates-formes définissent un moyen d'interaction que ce soit par un utilisateur ou tout simplement par l'intergiciel lui-même.

En résumé, l'alignement des opérations d'administrations dépend des concepts connus dans une technologie.

Notez cependant que dans SAM-CORE et donc dans SAM-CORE-RT le concept d'unité de déploiement n'existe pas, ni même celui de ressource. D'un point de vue purement service, celui-ci est disponible ou pas ; les notions d'installation, de validité ou de résolution sont des concepts de déploiement.

Le nombre d'états que peut avoir une matérialisation d'un service dans SAM-CORE est donc extrêmement limité. Un service est disponible ou ne l'est pas. Cependant selon la logique des groupes et d'instanciation, **un membre d'un groupe est disponible si et seulement si son représentant est disponible ;** par exemple une instance est disponible si son implémentation est disponible sur la plate-forme. Si **un représentant n'est plus disponible alors tous ses membres ne le sont plus non plus**, si une implémentation n'est plus disponible sur une plate-forme, alors ses instances sont détruites et donc elles ne sont plus disponibles. La Figure 71 résume les états et leurs transitions.

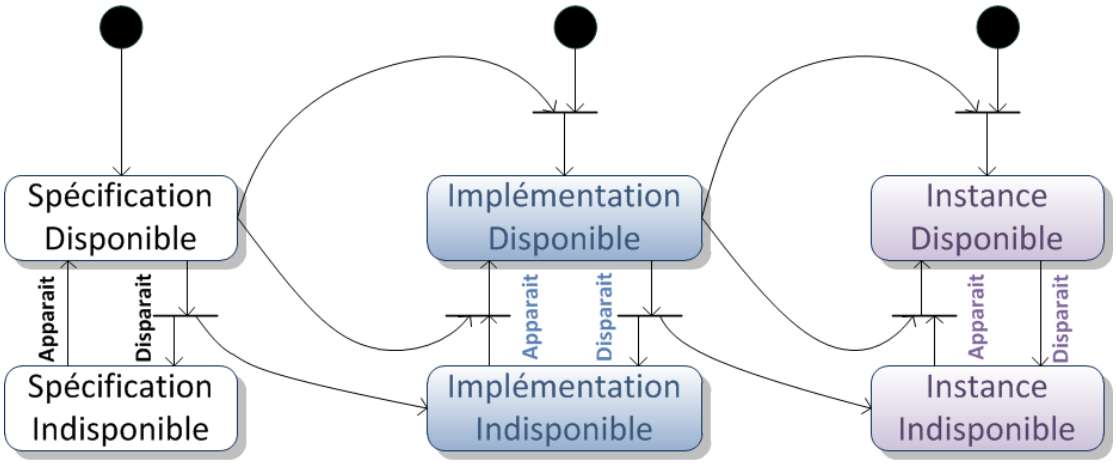


Figure 71 Cycles de vie d'un service SAM

5.2.4 Écart sémantique : Un modèle correct mais incomplet

Nous pouvons résumer en trois points les problèmes énoncés précédemment :

Un méta-modèle pour la description plus riche que le système étudié : Un problème majeur du méta-modèle SAM CORE est qu'il fournit des concepts qui ne sont potentiellement pas définis dans une technologie donnée. Par exemple, la plupart des plates-formes ne distinguent pas les concepts de spécification, d'implémentation et d'instance d'un service. De même le concept de dépendance explicite de service n'existe que dans les technologies de composants orientés service.

Des divergences sémantiques des concepts non-négligeables : Même si le même concept est disponible dans différentes technologies, il existe tout de même des différences – subtile ou non – qui les distinguent. Prenez le concept le plus évident : l'instance de service. Le nom, son cycle de vie, son identifiant, ses états, sa gestion des erreurs et beaucoup d'autres détails non négligeables peuvent avoir une sémantique ou une syntaxe différente d'une technologie à l'autre. De plus, fournir un identifiant vraiment unique pour un service, dans toutes les situations et pour toutes les technologies n'est pas aussi simple qu'il y paraît.

Des cycles de vie qui varient en fonction des préoccupations : chaque technologie SOC fournit (explicitement ou implicitement) un cycle de vie de service qui dépend des caractéristiques prises en charge par cette technologie.

Du point de vue des informations fournies par les plates-formes à service sous-jacentes, le modèle d'état d'exécution des services est valide : tout élément du modèle est « imageDe » un élément du système étudié ; mais il est aussi correct car tout élément identifié du système est « représentéPar » un élément du modèle. Le problème est que les technologies à service du système ne permettent pas d'identifier tous les éléments définis dans SAM-CORE, le modèle est correct car il fournit tous les éléments fournis par la technologie mais potentiellement incomplet vis-à-vis de l'espace d'exécution « réel ».

Par conséquent SAM-RT fournit un modèle d'état d'exécution des services « *Best effort* ». Ce modèle est obtenu uniquement à partir des environnements d'exécution non-SAM et ne tient pas compte de l'approche globale qui permet de compléter l'information fournie par les technologies sous-jacentes.

5.3 APPROCHE « TOP-DOWN »

L'approche proposée par SAM est une approche « Top-Down », c'est-à-dire guidée par les choix et les informations effectués et produits en amont. Il existe un environnement de développement de service spécifique à SAM. Dans cet environnement, il est possible soit de définir des services exploitant le SOA SAM : *service SAM* ; soir de définir des services selon un SOA existant enrichis par des méta-informations contenue dans leur unité de déploiement prise en charge par la SCM correspondante : *service « legacy » enrichi*.

Dans les deux cas les informations du service au développement sont transférées à l'exécution palliant ainsi l'écart sémantique dû au manque d'information. Dans le cas d'une SAM-RT où tous les services sont SAM ou enrichis, le modèle d'état des services de l'environnement d'exécution est une image parfaite (ou presque) de la « réalité ».

Dans les deux sous sections suivantes nous aborderons ces deux types de service.

5.3.1 Service SAM

L'environnement SAM est implémenté dans un contexte d'exécution Java. SAM-RT définit des annotations pour développer des services basés sur le SOA de SAM. Ces services sont néanmoins exécutés dans un environnement d'exécution concret ; par exemple : OSGi, iPOJO, Spring DM.

Un service SAM peut soit accéder aux registres via l'API :

```
import fr.imag.adele.am.LocalMachine;
import fr.imag.adele.am.Machine;
import fr.imag.adele.am.exception.ConnectionException;
import fr.imag.adele.sam.Implementation;
import fr.imag.adele.sam.Instance;
import fr.imag.adele.sam.Specification;
import fr.imag.adele.sam.broker.ImplementationBroker;
import fr.imag.adele.sam.broker.InstanceBroker;
import fr.imag.adele.sam.broker.SpecificationBroker;


public class SampleServiceImpl2SAM extends SampleServiceSpecificationSAM {

        public void sample() throws ConnectionException {
                Machine machine = LocalMachine.localMachine;
                if(machine!=null) {
                        // Registre de spécification
                        SpecificationBroker specificationBroker = (SpecificationBroker)
machine.getBrokerBroker().getBroker(SpecificationBroker.SPECIFICATIONBROKERNAME);
                        Specification specification =
specificationBroker.getSpecification("...");

                        // Registre d'implémentation
                        ImplementationBroker implementationBroker = (ImplementationBroker)
machine.getBrokerBroker().getBroker(ImplementationBroker.IMPLEMENTATIONBROKERNAME);
                        Implementation implementation =
implementationBroker.getImplementation("...");

                        // Registre d'instance
                        InstanceBroker instanceBroker = (InstanceBroker)
machine.getBrokerBroker().getBroker(InstanceBroker.INSTANCEBROKERNAME);
                        Instance instance = instanceBroker.getInstance("...");
                        // accès au service objet
                        Object serviceObject = instance.getServiceObject();
                }
        }
}
```

Nous voyons dans l'exemple ci-dessus qu'en passant par une variable statique, il est facile d'accéder aux différents registres ; cependant l'accès au registre nécessite plusieurs de lignes de code

qui sont parasites vis-à-vis du code métier. C'est pourquoi SAM-RT définit un ensemble d'annotations ayant pour but de simplifier l'utilisation du SOA SAM.

```
import fr.imag.adele.am.exception.ConnectionException;
import fr.imag.adele.sam.Implementation;
import fr.imag.adele.sam.Instance;
import fr.imag.adele.sam.Specification;
import fr.imag.adele.sam.annotations.specification;
import fr.imag.adele.sam.annotations.implementation ;
import fr.imag.adele.sam.annotations.instance;

public class SampleServiceImplSAM extends SampleServiceSpecificationSAM {

        @specification (id="...")
        private Specification specification;

        @implementation (id="...")
        private Implementation implementation;

        @instance (id="...")
        private Instance instance;

        public void sample () throws ConnectionException {
                // accès au service objet
                Object serviceObject = instance.getServiceObject();
                //...
        }

    }
```

En résumé un service SAM est un service utilisant le SOA de SAM mais s'exécutant sur une plate-forme à service « concrète ».

5.3.2 Service « legacy » enrichi

Les services « legacy » sont des services dont l'implémentation est propre à une technologie et par conséquent ils ne font pas référence à SAM. Cependant il est possible de leur adjoindre des métadonnées dans le but de combler l'écart sémantique entre la technologie cible et SAM CORE. Le format des métadonnées, comme la manière de les lier aux services, sont des aspects spécifiques à la SCM qui prend en charge la technologie cible.

5.4 SYNTHESE

Une technologie à service est utilisée pour un problème précis, par exemple, si nous voulons faire interopérer des systèmes hérogènes nous utilisons des Web-Services, si nous voulons communiquer avec des équipements domestiques nous utiliserons plutôt des technologies comme UPnP ou DPWS, ou si nous voulons une plate-forme d'exécution à la carte comme pour Jonas 5[18], nous utliserons des technologies comme OSGi. Nous voyons à travers ces exemples que l'utilisation de technologies à service dépend avant tout du besoin du domaine métier. Or, les besoins actuelles tendent à l'utilisation simultanée de différentes technologies. Cette multiplication des technologies à service dans un même domaine devient problèmatique car les technologies ne partagent pas leurs connaissances comme l'identification, la syntaxe ou la sémantique. Au final, le défi ne vient plus de la résolution d'un problème par une application, mais de la complextié et la non compatibilité des technologies utilisées pour définir

[18] http://wiki.jonas.ow2.org/xwiki/bin/download/JOnASDays/JOnAS+Day+5_1/Matin_Partie_%232_Architecture_OSGi-v1.0.pdf

l'application. C'est pourquoi, un des buts de l'approche SAM est d'intégrer les différentes technologies à service. En résumé SAM-CORE-RT est un SOA qui à pour but d'unifier différentes technologies à service dans un même modèles, c'est-à-dire qu'il ne possède pas d'exécution propre, ou plutôt, que son espace d'exécution est l'union des espaces d'exécution des technologies qu'il intègre.

SAM-CORE-RT est architecturé principalement en deux niveaux :

- le niveau générique qui le méta-modèle pour fournir le modèle d'état d'exécution des services.
- le niveau spécifique – représenté par les SCM – qui contient les règles d'indentification, de réprésentation et inversement celles de matèrialisation. Les SCM assurent la correction et la validité entre les différents systèmes et le modèle d'état.

Cependant, SAM CORE est un méta-modèle composant orienté service sémantiquement plus riche que la plupart des approches orientées service. Par conséquent, en fonction de la technologie intégrée, il existe de forts écarts sémantiques qui se traduisent généralement par des informations incomplétes sur un service et/ou l'impossibilité de manipulé son cycle de vie et donc de modifier l'architecture globale. Toutefois, SAM-CORE-RT fournit au moins, sinon plus, les informations que la plate-forme d'origine. De ce point de vue SAM-CORE-RT fournit une représentation « best effort » des technologies à service, et puisqu'il fournit au moins les informations fournies par la technologie cible SAM-CORE-RT peut se substituer à celle-ci.

Il faut noter toutefois que l'intégration dépend au final de l'implémentation des SCM. Une même technologie peut être intégrer différemment dans SAM en fonction des choix d'implémentation de la SCM. Par conséquent, de la même manière que SCA, il serait judicieux de spécifier pour chaque technologie les règles d'identification, de représentation et de matérialisation. L'interprétation des méta-données pour les services « legacy » enrichis, ou le support de service SAM pour cette technologie reste à la discrétion de la SCM.

6. EXTENSIBILITE DE L'ENVIRONNEMENT D'EXECUTION

Chaque technologie a pour but (souvent implicite) de résoudre un problème particulier, et s'intéresse à des aspects particuliers. Ces aspects peuvent être requis par la technologie ou peuvent résoudre des préoccupations supplémentaires. Reprenons comme exemple la technologie OSGi : celle-ci est un système de modules dynamiques pour Java™. La plate-forme à Service OSGi fournit un environnement pour l'intégration et donc pour le développement logiciel. Cette brève description est extraite de la page Web « *Technology* »[19] du site de l'OSGi Alliance. Nous voyons que le principal but d'OSGi est de fournir un environnement de modules dynamiques pour l'intégration logicielle. Il est donc normal de retrouver dans son cycle de vie (cf. Figure 69) des préoccupations de déploiement de module.

Dans l'approche SAM une application peut être définie par un orchestrateur où les services utilisés sont importé dans *SAM-RT* via une *Service Concrete Machine* pour les *Web-Services* [PE09]; dans ce cas la préoccupation du déploiement de service est optionnelle. L'approche SAM a avant tout pour but de définir un environnement minimal et nécessaire au développement et à l'exécution d'applications orientées service dynamiques.

Dans cette section nous nous intéresserons à la séparation des préoccupations dans SAM-RT. Dans un premier temps nous allons rappeler les différents aspects natifs à la SAM-RT, puis nous allons classifier les préoccupations afin de fournir un mécanisme d'intégration dans SAM-RT pour chacune de ces classes.

6.1 SEPARATION DES PREOCCUPATIONS DANS SAM-RT

Le but de cette thèse est de fournir un environnement d'exécution homogène de services hétérogènes pour garantir l'exécution d'applications dynamiques : SAM-RT. Ce but est décomposable en deux sous préoccupations :

- fournir un environnement d'exécution homogène permettant l'intégration de technologies à service hétérogènes potentiellement distribuées et dynamiques : SAM-CORE-RT ;
- fournir un environnement réparti spécialisable : l'*Abstract Machine* (AM).

SAM-CORE-RT est une spécialisation de l'AM fournissant l'environnement d'exécution orienté service pouvant être étendu à d'autres préoccupations : déploiement, dépôt, application, sécurité… via le registre d'extension de l'AM. Une préoccupation est intégrée dans SAM-RT comme une extension de l'*AM*. Dans l'approche orientée service il existe de nombreux problèmes / préoccupations pouvant se recouvrir partiellement. Nativement l'AM ne cherche pas à résoudre la concurrence entre deux extensions ; comme nous l'avons dit dans la section 4.2 de ce chapitre, l'AM est spécialisable pour un contexte d'exécution donné ; comme pour l'intégration des SCM, on n'ajoute jamais au hasard une préoccupation, c'est à l'administrateur (humain ou machine) de prendre en charge la composition de l'AM est donc de résoudre les conflits liés au recouvrement.

Nous avons identifié trois types d'extensions pour SAM-RT :

- Les extensions qui ont pour but d'intégrer au modèle d'état les préoccupations des technologies sous-jacentes comme par exemple le déploiement : *Integration Runtime* ;
- Les extensions qui ont pour but d'ajouter au modèle d'état une préoccupation spécifique à l'AM, par exemple un dépôt d'unité de déploiement spécifique au service SAM : *Runtime* ;

[19] http://www.osgi.org/About/Technology

- Les extensions qui ont pour but d'agir sur le modèle d'état pour diriger ou garantir un aspect comme l'exécution d'une application : *Manager*.

Dans cette section nous ne détaillerons pas les *Managers*, nous y reviendrons dans la section 7 de ce chapitre ainsi que dans le Chapitre 4 - .

6.2 GENERALISATION DE LA MECANIQUE DU *RUNTIME*

Dans notre approche, nous nous concentrons sur les activités du génie logiciel qui ont des relations directes avec la phase d'exécution ou qui sont en interaction avec l'environnement d'exécution. Les *Runtimes* et les *Integration Runtimes* étendent le méta-modèle SAM-CORE avec de nouveaux concepts, de nouvelles propriétés ainsi que de nouveaux états dans le diagramme de transition d'état (STD) pour chaque concept.

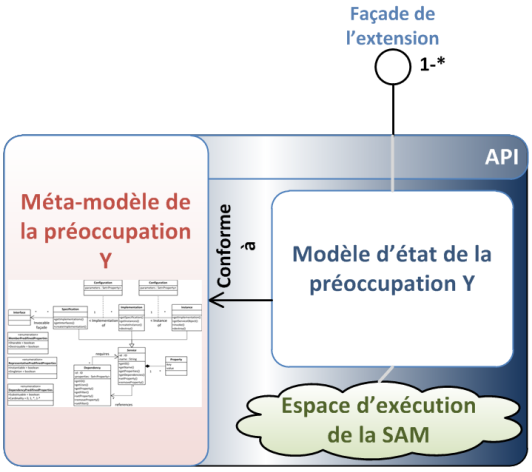


Figure 72 Architecture d'un *Runtime*

La Figure 72 expose l'architecture typique d'un *runtime*. Un *runtime* a pour but de prendre en charge une préoccupation – dans la Figure 72 c'est la préoccupation Y – est de fournir un modèle d'état de cette préoccupation. Ce modèle est accessible via une ou plusieurs façades de l'API exposées. Ces façades sont enregistrées comme des extensions dans l'AM. Certaines de ces préoccupations dépendent des technologies sous-jacentes à la SAM. Ces préoccupations nécessitent donc l'intégration des technologies existantes dans un modèle homogène. Nous retombons dans la même problématique que pour SAM-CORE, nous réutilisons le même mécanisme : les SCM, pour résoudre ce problème.

Les *runtimes* intégrant les préoccupations des technologies sous-jacentes s'appellent des *runtimes* d'intégration (*Integration runtimes*). Cependant, en plus d'intégrer les préoccupations des différentes technologies, ils peuvent posséder un espace propre : par exemple, la préoccupation de déploiement de service nécessite l'intégration des plates-formes sous-jacentes mais nous pouvons aussi définir une unité de déploiement propre aux services SAM. Par conséquent, un *runtime* d'intégration peut avoir les deux architectures définies dans la Figure 73.

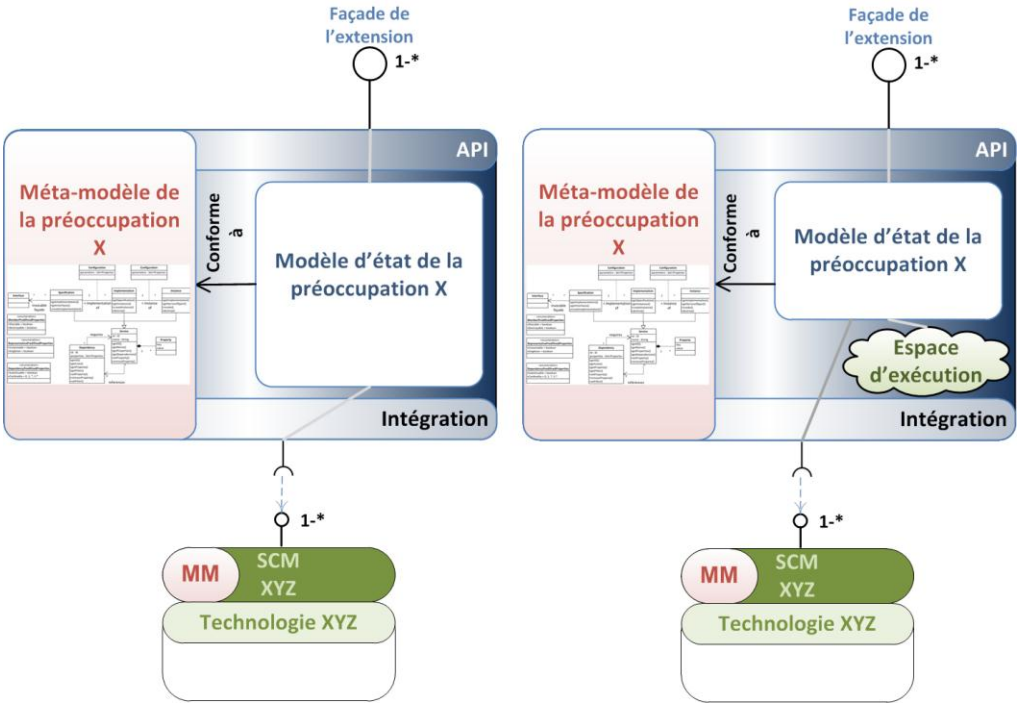


Figure 73 Architectures d'un *Runtime* d'intégration

6.3 SYNTHESE

Nous avons généralisé le concept de *runtime*. Un *runtime* :

- a pour but de prendre en charge à l'exécution un point de vue particulier de l'approche orientée services. Une vue est constituée d'un méta-modèle qui décrit les concepts associés, leurs propriétés, et un ensemble d'états pour chaque concept du méta-modèle pour les STD.
- peut étendre un ou plusieurs autres *runtimes*, il peut ajouter des propriétés sur les concepts d'autres *runtimes*.
- peut avoir un ensemble de *Service Concrete Machine* visant à intégrer le point de vue d'une technologie SOA existante. Un *runtime* basé sur des SCM est appelé *Integration runtime*.
- maintient un modèle de l'État correspondant à ce point de vue.
- fournit une API correspondant à ce point de vue. Cette API doit fournir au moins les actions pour modifier les états définis dans le schéma de transition d'état.

Les *runtimes* ont à peu près le même niveau d'abstraction, leur but est d'étendre le méta-modèle SOA avec de nouveaux concepts, de nouvelles fonctionnalités ou de nouvelles propriétés. Ce mécanisme d'extensibilité peut être considéré comme une intégration horizontale, par opposition à l'intégration verticale qui consiste à intégrer des technologies existantes dans un modèle commun situé à un niveau supérieur d'abstraction -.

De même nous avons généralisé le concept d'une SCM que nous nommerons IM pour *Integration Machine*. Une IM est un médiateur entre un système (réel) et un *runtime* abstrait. Une IM:

- intégre /modélise les éléments identifiés par le méta-modèle associé à son *runtime*.
- assure l'alignement sémantique et syntaxique entre le système et le modèle du *runtime*.
- doit garantir la correction du modèle au système.
- doit garantir la validité du système au modèle dans le cas où le modèle du *runtime* est manipulable.

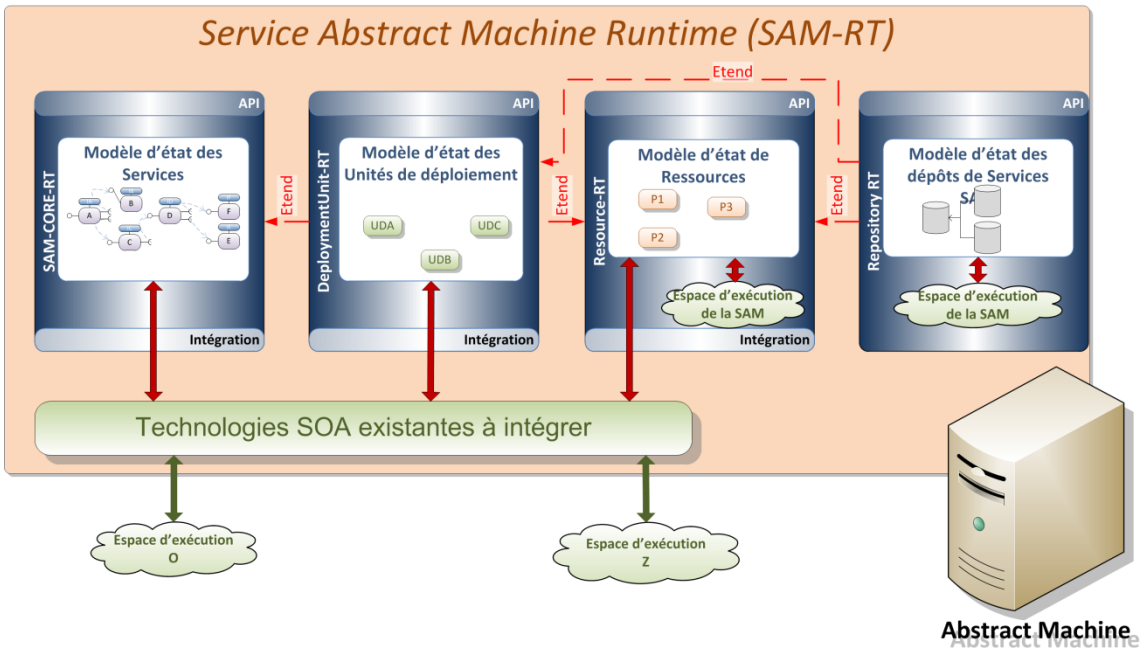


Figure 74 Exemple d'une SAM-RT étendue

La Figure 74 montre un exemple d'une SAM-RT étendue par trois préoccupations : les unités de déploiement, les ressources et les dépôts. Cet exemple sera détaillé dans le Chapitre 4 -

7. SYNTHESE : MODELE D'ETAT

Le but de cette thèse est de concevoir un environnement d'exécution permettant l'adaptabilité des applications basées sur des services hétérogènes. L'une des premières difficultés est la définition même d'un service. Il n'y a pas de consensus si ce n'est le fait qu'un service est défini par une définition abstraite sur laquelle sont basées la sélection et la communication. Le but était de masquer les notions d'instance et d'implémentation permettant ainsi le faible couplage, le masquage de la distribution et de l'hétérogénéité. Cependant, cette abstraction a perdu trop d'information du système nécessaire à l'adaptabilité. En effet, dans un tel contexte d'exécution (cf. *Web-Service*) l'adaptabilité est réduite qu'à la substitution d'un service par un autre ; ou plus simplement il n'est pas possible d'agir sur le cycle de vie des services et donc il n'est pas possible de modifier l'architecture de l'application à l'exécution. Cependant, l'approche à composant fournit ces informations, par contre la flexibilité est rarement prise en compte et l'architecture de l'application est très souvent statique.

Le méta-modèle de service défini dans cette thèse : *SAM CORE*, est issue de **l'approche composant orienté service** (cf. Chapitre 2 - section 4.4). Cette approche – définie par H. Cervantes [CerT04] – est basée sur la convergence de l'approche à service et celle à composant où es interactions entre les composants du système suivent l'approche à service. Ce méta-modèle distingue clairement les différentes matérialisations d'un service que sont la spécification, l'implémentation et l'instance, et ce, quel que soit la phase du cycle de vie. L'information est continue dans le cycle de vie de l'application.

A l'exécution, avoir les concepts d'instance et de dépendance permet de définir l'architecture de l'état d'exécution des services de la plate-forme. Le modèle d'état de l'architecture est donc un **modèle descriptif** de service. De plus, avoir le concept d'implémentation permet d'agir sur le cycle de vie d'une instance de service et donc d'agir sur l'architecture. De ce point de vue le **modèle** d'état devient **prescriptif** car il permet la modification de l'architecture.

De ce fait, l'approche prise par cette thèse est en pleine concordance avec le point de vue de R. France et B. Rumpe :

> *"Attempts at building complex software systems that dynamically adapt to changes in their operating environments has led some researchers to consider the use of models during runtime to monitor and manage the executing software. […]*
>
> *We envisage that MDE research on runtime models will pave the way for the development of environments in which change agents (e.g., software maintainers, software-based agents) use runtime models to modify executing software in a controlled manner. The models act as interfaces that change agents can use to adapt, repair, extend, or retrofit software during its execution. In our broad vision of MDE, models are not only the primary artifacts of development, they are also the primary means by which developers and other systems understand, interact with, configure and modify the runtime behavior of software."*
>
> Extrait de [FR07]

Le méta-modèle SAM-CORE fournit les concepts nécessaires pour l'adaptation dynamique des applications.

Ce méta-modèle est pris en charge par la plate-forme d'exécution SAM-RT. Cependant, les besoins actuelles tendent à l'utilisation simultanée de différentes technologies. La multiplication des technologies à service dans un même domaine devient problèmatique car les technologies ne partagent pas leurs connaissances comme l'identification, la syntaxe ou la sémantique. Au final, le défi informatique ne vient plus de la résolution d'un problème par une application, mais de la complextié et la non compatibilité des technologies utilisées pour définir l'application. C'est pourquoi, un des buts de

l'approche SAM est d'intégrer les différentes technologies à service. Pour cela, SAM-CORE-RT intègre différentes technologies à service dans un même modèles. Cependant, SAM CORE est sémantiquement plus riche que la plupart des approches orientées service. Il peut donc exister de forts écarts sémantiques. Ces écarts se traduisent généralement par des informations incomplétes sur un service et/ou l'impossibilité de manipuler son cycle de vie et donc de modifier l'architecture globale. Toutefois, SAM-CORE-RT fournit au moins, sinon plus, les informations que la plate-forme d'origine. De ce point de vue SAM-CORE-RT fournit **une représentation « best effort »** des technologies à service, et puisqu'il fournit au moins les informations fournies par la technologie cible SAM-CORE-RT peut se substituer à celle-ci. Il est toutefois possible de définir des **services SAM** qui sont donc sémantiquement fidèles aux méta-modèle SAM-CORE. De ce point de vue SAM-CORE-RT fournit **une représentation « exacte »** de l'état d'exécution des services de la plate-forme.

Dans tous les cas, SAM-CORE-RT possèdent toutes les propriétés d'un **SOA dynamique**, qui **fournit un mécanisme d'intégration des technologies SOA**. Cette intégration, d'une part, **caractérise et représente un service dans un modèle homogène**, et d'autre **intègre les fonctionnalités des plates-formes de la gestion des cycles de vie** des services.

Cependant, il ne faut pas réduire l'adaptabilité d'une application seulement aux aspects de disponibilité dynamique des services, de même la réponse à ce genre d'évènement n'est pas la sélection d'une nouvelle instance de service ou la création d'une nouvelle instance. Cette adaptabilité doit pouvoir dépendre d'une multitude de préoccupations du contexte d'exécution et pouvant se résoudre d'une multitude de façons dépendamment du métier. C'est pourquoi la plate-forme SAM-RT peut être étendue par des préoccupations. Le but est d'étendre le méta-modèle SOA avec de nouveaux concepts, de nouvelles fonctionnalités ou de nouvelles propriétés. Ce mécanisme d'extensibilité peut être considéré comme une intégration horizontale, par opposition à l'intégration verticale qui consiste à intégrer des technologies existantes dans un modèle commun situé à un niveau supérieur d'abstraction. Nous généralisons donc les concepts de *runtimes* et de *SCM* à toutes préoccupations. Cette intégration horizontale (extensibilité) est prise en charge par la plate-forme d'exécution (cf. la section 4.2 de ce chapitre). **La plate-forme définit un mécanisme d'intégration de nouvelles préoccupations du contexte d'exécution**.

En résumé la plate-forme SAM-RT fournit un ensemble de modèles d'état qui ont pour but d'abstraire et de modéliser des points de vue (préoccupation) de l'environnement d'exécution. Les modèles d'états ne sont pas uniquement descriptifs, ils fournissent aussi les opérations permettant la manipulation de l'environnement d'exécution pour une préoccupation donnée. **Ces modèles d'état équivalent aux sondes et aux actionneurs nécessaires à la mise en place d'une boucle de contrôle/décisionnel.**

Dans l'approche SAM, l'adaptabilité de l'application est implantée par un manageur de l'application. Le manageur d'application joue le rôle du bloc de contrôle/décisionnel. Comme montrer par la Figure 75, un manageur dépend des sondes fournies par les modèles d'état (descriptif) pour analyser si l'état courant est conforme à ce qu'il souhaite. Si l'état courant nécessite une adaptation alors le manageur planifie les opérations sur les différents actionneurs fournis par les modèles d'état (prescriptif).

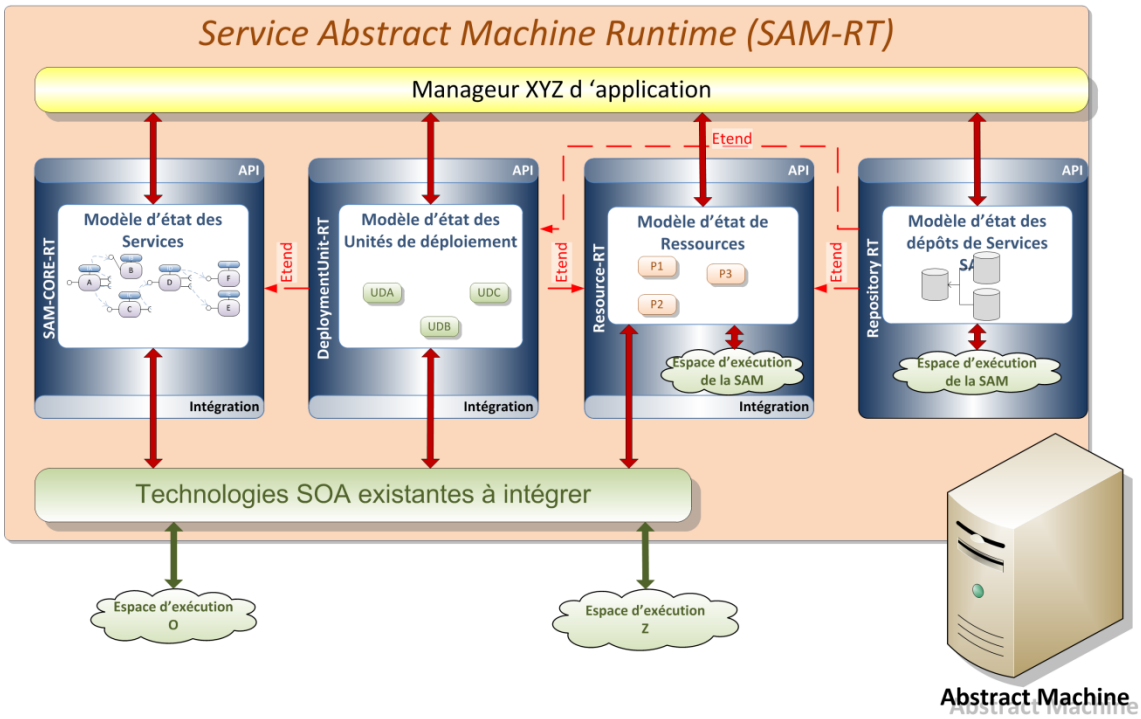


Figure 75 Bloc de contrôle : manageur d'application

En résumé SAM-RT est un environnement d'exécution fournissant les mécanismes nécessaires à l'adaptabilité des applications basées sur des services dynamiques et hétérogènes.

Chapitre 4 - Évaluation et Validation

Dans le chapitre précédent, nous avons défini un environnement réparti d'exécution de services hétérogènes autorisant la modification de l'architecture courante à l'exécution. Cet environnement se décompose en deux principaux blocs : l'environnement réparti et l'environnement d'exécution à service. Dans ce chapitre nous allons nous intéresser à la validation des différents mécanismes mis en place pour un tel environnement.

Nous allons dans un premier temps décrire les différents projets internes et externes à l'équipe qui se sont basés sur le prototype résultant de cette thèse ; fournissant ainsi un ensemble d'exemples et de scénarii pour les différents aspects. Nous commencerons par valider les mécanismes de l'environnement réparti qui sont : la découverte, la communication et les événements. Puis nous nous intéresserons aux mécanismes de l'environnement d'exécution à service qui sont : l'intégration de technologies à service et l'extensibilité du contexte d'exécution à d'autres préoccupations.

1. PROJETS

Cette thèse a été réalisée dans le cadre du projet Européen SEMbySEM[20] dans le but de fournir un environnement d'exécution. Cependant SAM-RT a servi aussi dans deux autres projets de l'équipe : SELECTA et FOCAS. Dans cette section, nous allons décrire brièvement ces différents projets et nous intéresser à l'utilisation de SAM-RT dans leur contexte.

1.1 SEMBYSEM

Le prototype de l'environnement a été développé en partie dans le cadre du projet Européen SEMbySEM (ITEA2). Le but de ce projet est de réaliser une médiation sémantique entre des concepts métiers pour la supervision et des concepts techniques définis en termes de services ; d'où le nom du projet : *SErvice Management by SEMantique.* L'idée sous-jacente est de fournir une visualisation d'un système la plus proche possible du domaine métier.

Architecture

L'architecture définie dans le projet est relativement simple. Il y a trois modules qui communiquent au travers de protocoles partagés (cf. Figure 76). Ces modules sont : la visualisation, le CORE et la Façade.

- Comme son nom l'indique, le module de visualisation contient la partie de visualisation. Ce module contient l'ontologie du domaine métier utilisée pour la visualisation. Le module peut envoyer et recevoir des messages du système.
- Le module CORE effectue l'alignement sémantique des messages échangés entre la visualisation et le système administré.

[20] Site du Projet : http://www.sembysem.org/

- Le module Façade masque l'infrastructure du système au CORE. Il s'interface pour aligner les protocoles de communication et les syntaxes utilisés dans le système avec ceux utilisés par le CORE, et vice-versa.

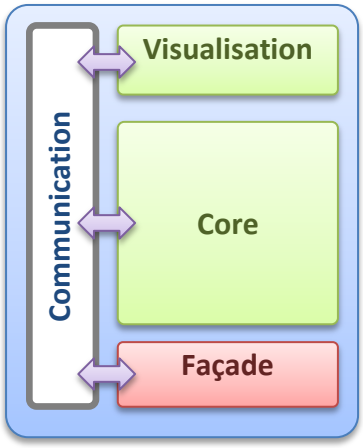


Figure 76 Architecture simplifiée du projet ITEA SEMbySEM

Dans ce projet, SAM-RT fournit une implémentation de la Façade. En effet, SAM-RT permet de définir facilement une infrastructure répartie et dynamique permettant la remontée de données. Dans ce projet, les services et les équipements du système que l'on souhaite intégrer sont pris en charge par des proxies spécifiques qui remontent les données au travers de l'infrastructure répartie jusqu'à la SAM-RT qui fait office de *Façade*. Cette dernière retransmet alors les messages dans un format et un protocole de communication supportés par le module CORE.

Démonstrateur

Le choix du domaine métier du démonstrateur est issu d'un des cas d'utilisation définis dans SEMbySEM, qui est celui de la remontée de données des équipements d'un réseau ferroviaire.

Le cas d'utilisation de notre démonstrateur est la supervision des trains de fret. Le but est qu'un administrateur puisse connaître à tout moment la localisation (GPS), la composition (wagon et locomotive) et le contenu d'un train (marchandise). La Figure 77 schématise l'infrastructure du démonstrateur :

- Un wagon possède une passerelle sur laquelle sont branchés des équipements. Il existe deux types de wagon de marchandises : les classiques et les réfrigérés. Dans les deux cas, le wagon contient un lecteur RFID qui fait un inventaire automatique des marchandises (tag RFID). De plus, un wagon réfrigéré contient un capteur de température.
- Une locomotive possède une passerelle sur laquelle sont branchés des équipements. Elle possède un GPS et un capteur de vitesse.
- Un train est composé d'une locomotive et d'un ensemble de wagons. L'instance de ce concept est localisée sur la passerelle de la locomotive.
- La passerelle de la locomotive est reliée à la Façade. La Façade retransmet les messages dans le format et le protocole de communication de la cible (CORE ou locomotive).
- La Façade est connectée au CORE.

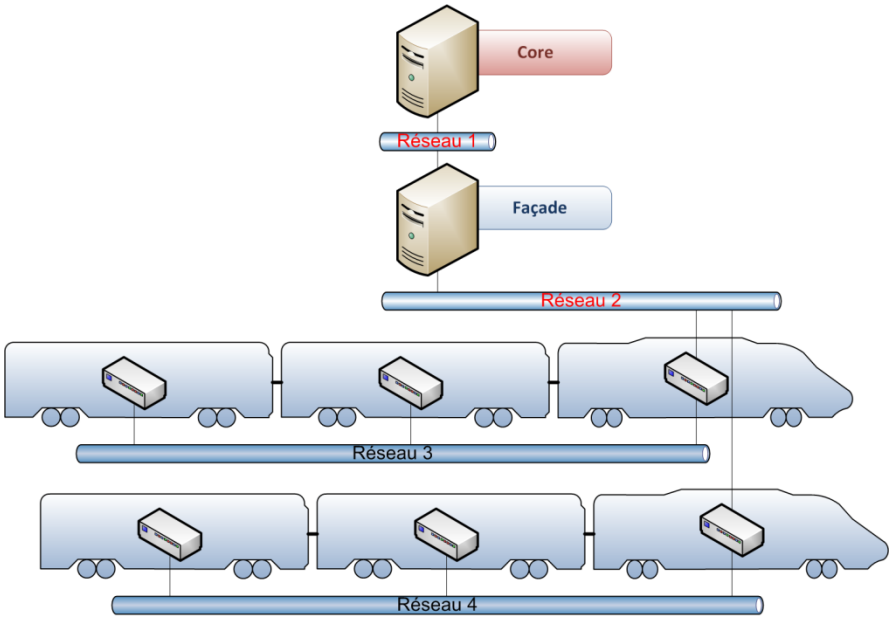


Figure 77 Schéma du démonstrateur pour SEMbySEM

Nous voyons dans le schéma de la Figure 77 que plusieurs réseaux coexistent. En effet, il y a peu de chance que les wagons soient connectés en Wifi à une locomotive, mais plutôt via un bus de terrain. De même, une locomotive ne sera pas connectée en filaire au serveur centralisé de Façade.

Comme nous l'avons dit précédemment, SAM-CORE-RT ne définit pas de niveau application. Dans notre cas l'application se construit de manière programmatique. Elle est constituée de trois modules administrables implémentés par des composants orientés service, d'un ensemble de proxies spécifiques pour capteurs, et d'un module façade qui fait l'alignement protocolaire et syntaxique.

- Le module Wagon est un composant orienté service SAM qui dépend de zéro ou plusieurs équipements : dans notre cas un lecteur RFID, et potentiellement un capteur de température. Le service Wagon implémente une interface d'objet administrable (*ManageableObject*) SEMbySEM. Il remonte les données fournies par ces capteurs via le mécanisme d'évènement de l'AM.

```
package org.sembysem.sam.api;

public interface ManageableObject {

    public String getInstanceID();
    public String getConceptID();
    public String getAMTopic();


}
```

- Le module Locomotive est un composant orienté service SAM qui dépend de zéro ou plusieurs équipements : dans notre cas un GPS et un capteur de vitesse. Le service Locomotive implémente une interface d'objet administrable (*ManageableObject*) SEMbySEM. Il remonte les données fournies par ces capteurs via le mécanisme d'évènement de l'AM.
- Le module train est un composant orienté service SAM qui dépend d'une locomotive et de zéro ou plusieurs wagons. Pour cela il recherche localement via les registres SAM le service Locomotive. Il recherche les AM distantes sur les réseaux, et lorsqu'il trouve une AM, il recherche activement (requête) et passivement (souscription aux notifications d'instance) des objets administrables de type Wagon.
- Le module façade est un composant orienté service SAM qui recherche les AM sur les réseaux. Pour chaque AM trouvée, il recherche activement et passivement les objets administrables et

souscrit à leurs événements. Lorsqu'il reçoit un évènement, il créé un message JMS (dans le cas présent ActiveMQ) contenant le message sérialisé (JAXB) dans le format SEMbySEM, puis l'envoi vers un serveur JMS configuré statiquement.

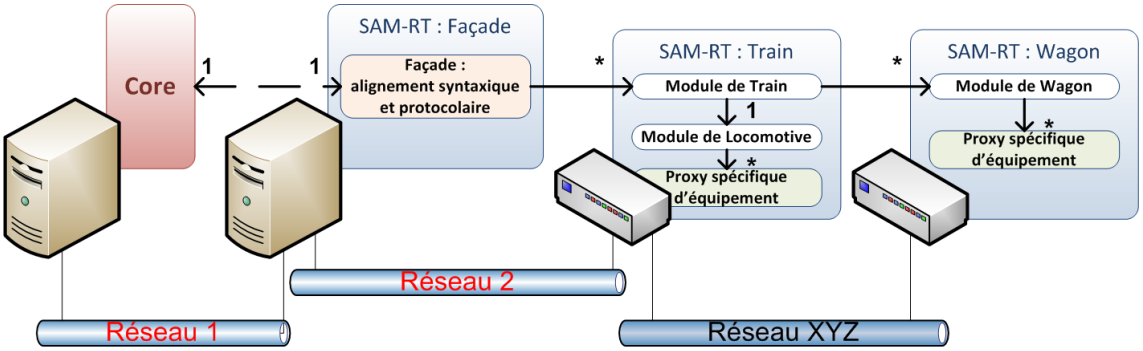


Figure 78 Schéma logique du démonstrateur

La Figure 78 résume les interactions et les dépendances de chacun de ces éléments.

L'utilisation de SAM-RT

Dans le projet SEMbySEM, l'utilisation de SAM-RT trouve pleinement sa place. Elle permet de construire une application au-dessus d'un environnement réparti dynamique. Dans le démonstrateur, les éléments de l'application sont implémentés en tant que service SAM. La communication entre les éléments est basée soit sur des appels synchrones soit sur des messages asynchrones via le mécanisme d'événement de l'AM.

L'infrastructure proposée permet d'un côté de définir des services métiers au plus près des éléments à administrer, et de l'autre une création dynamique de l'application par l'apparition des éléments. Notons que si l'on souhaitait passer à l'échelle, nous pourrions déployer sur les nœuds intermédiaires des services d'agrégation ou de compression dans la remontée de données.

Scénario du démonstrateur

Le scénario du démonstrateur est basé sur deux trains :

1. Le module façade est activé et connecté au CORE.
2. Les passerelles des deux locomotives sont activées.
3. La passerelle de Wagon d'un des trains est activée.
4. Le deuxième Wagon est activé.
5. Le deuxième Wagon est déconnecté et on le connecte sur le réseau de la deuxième locomotive.
6. Une ou plusieurs étiquettes RFID représentant les marchandises peuvent être placées ou enlevées d'un ou plusieurs lecteurs RFID.

1.2 VALIDATEUR POUR FOCAS

Le projet FOCAS [PedT09] [PE08] [PDE09] pour *Framework for Orchestration, Composition and Aggregation of Services* en anglais se place à l'intersection entre la technologie des *workflow*, l'approche à service et l'ingénierie dirigée par les modèles, afin de fournir le support pour la création et l'exécution d'applications orientées procédé. Il utilise le modèle de procédé de la technologie de *workflow* comme élément central et structurant. Il utilise l'approche orientée service comme technologie pour l'implémentation de ses procédés, afin d'obtenir la flexibilité fournie par les propriétés de faible couplage et de liaison retardée. Et finalement, il utilise l'ingénierie dirigée par les modèles pour fournir

un cadre pour la mise en place effective du projet, en utilisant les modèles comme artefacts de premier ordre dans la construction des applications orientées procédé.

L'utilisation de SAM-RT

Une des implémentations de FOCAS a été portée sur SAM-RT pour la raison suivante :

> *« Une propriété de la machine SAM est qu'elle a la capacité de cacher l'hétérogénéité des implémentations des services. L'observateur n'a pas connaissance de la technologie utilisée par l'instance du service retournée. Du point de vue de l'observateur, tous les services sont des classes Java qui implémentent les interfaces utilisées pour décrire le service requis. La SAM est chargée alors d'implémenter le protocole utilisé pour communiquer avec la vraie instance du service, et de fournir un proxy à l'observateur. Actuellement les technologies OSGi, UPnP, DPWS et services Web sont supportés par la SAM. »*
>
> Extrait de [PedT09]

En résumé l'orchestrateur FOCAS utilise SAM-RT comme un niveau d'abstraction cachant la technologie sous-jacente. Il ne cherche pas à déployer, à administrer ou à faire inter-opérer des services entre eux ; il l'utilise pour l'intégration de services dans son espace d'exécution.

1.3 DEMONSTRATEUR POUR LE PROJET SELECTA

Le projet SELECTA [DieT10] [EDSG09] [EDL10] a pour objectif de définir avec un niveau d'abstraction assez élevé, les concepts de composite et d'application. L'intention est de fournir des environnements logiciels spécialisés en fonction des activités associées à chaque phase du cycle de vie dans le but de faciliter le développement d'applications de la conception à l'exécution.

L'utilisation de SAM-RT

Dans [DieT10] M Idrissa Dieng utilise SAM-RT de la façon suivante :

> *« Dans notre système, nous définissons le concept de composite comme une extension de la machine SAM […]. L'extension Composite SELECTA que nous proposons est l'une des extensions possibles du noyau de base SAM CORE […]. L'infrastructure d'exécution des composites, nommée SELECTA RUNTIME, étend la plateforme d'exécution SAM et s'appuie sur les fonctionnalités fournies par ce dernier. »*
>
> Extrait de [DieT10]

SELECTA propose un environnement homogène allant de la conception à l'exécution d'applications basées sur le méta-modèle SAM-CORE (cf. Figure 79).

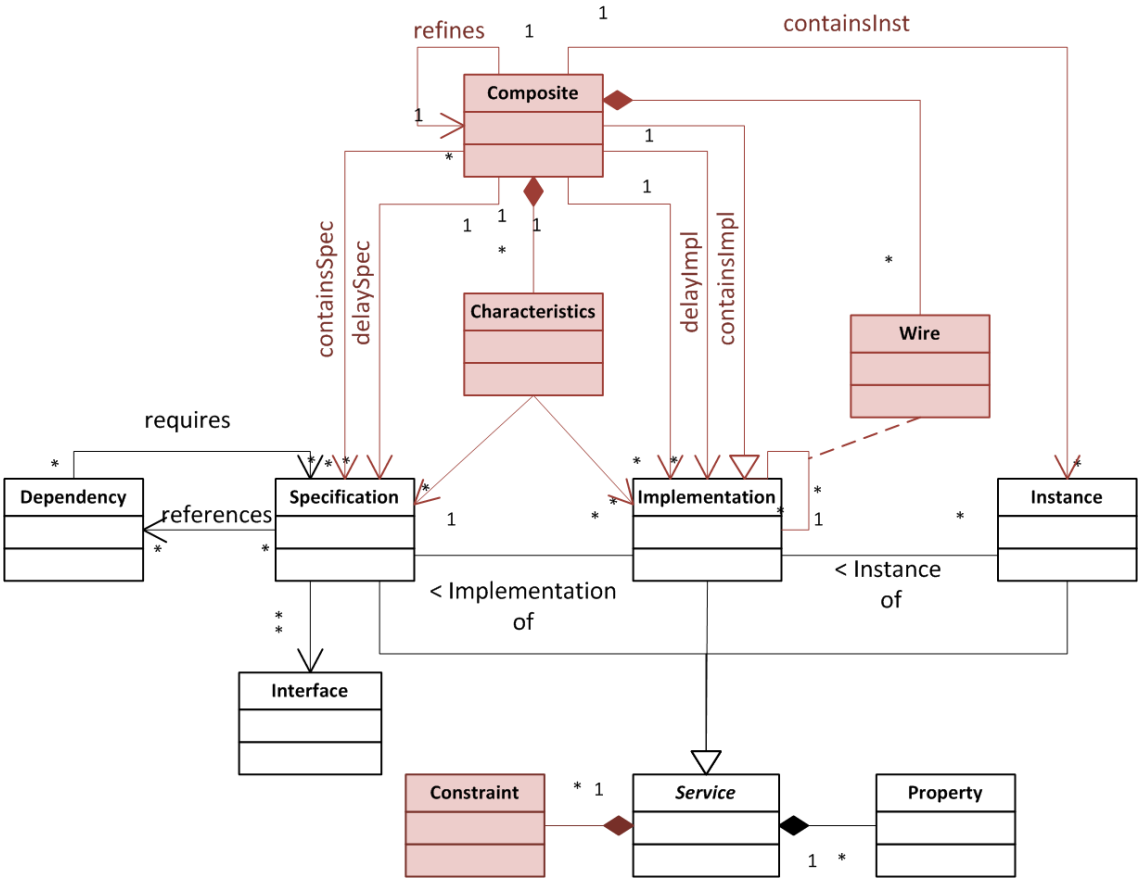


Figure 79 Méta-Modèle du Composite SELECTA pour SAM-CORE

SELECTA runtime est une extension de SAM-CORE-RT ajoutant les concepts pour la définition de composites.

Lors de la soutenance de thèse de M Idrissa Dieng, celui-ci a effectué la démonstration suivante : il a défini un composite basé sur des services SAM dans la technologie iPOJO à l'aide de son environnement de développement spécialisé. Il a ensuite transféré le modèle du composite à un manageur d'application. Le manageur d'application s'exécute sur une AM et dépend de SAM-CORE-RT et *DeploymentUnit-RT* (cf. section 0 de ce chapitre).

Le manageur d'application analyse le modèle cible et agit sur les deux *runtimes* pour initialiser l'application conformément au modèle.

2. ENVIRONNEMENT REPARTI : *ABSTRACT MACHINE*

L'environnement réparti est construit par l'interconnexion des *Abstract Machines* (AM). L'AM prend en charge trois aspects : la distribution, la spécialisation de l'environnement et les notifications. L'AM est implémentée au-dessus de la plate-forme OSGi et utilise la technologie iPOJO.

Dans cette section, nous aborderons chacun de ces aspects avant de conclure sur un ensemble de métriques.

2.1 DISTRIBUTION

La distribution est composée de deux aspects : la découverte et la communication entre machine.

Comme expliqué dans le Chapitre 3 - section 4.2.1, les protocoles de découverte sont découplés du module de gestion des machines. De même, l'implémentation de l'AM fait un effort particulier sur le découplage du protocole de communication de son utilisation. En d'autres termes, l'AM base sa découverte et sa communication sur des abstractions et non sur des technologies précises.

L'implémentation de l'AM permet de prendre en charge plusieurs interfaces réseaux. De plus, elle est capable de mettre à jour dynamiquement – c'est-à-dire sans interruption de l'AM – la liste des adresses de communication.

2.1.1 Communication

Le mécanisme de communication est simple. Chaque AM expose son interface d'administration (*fr.imag.adele.am.Machine.java*), le registre de machines (*fr.imag.adele.am.broker.MachineBroker*) ainsi que son registre d'extensions (*fr.imag.adele.am.broker.BrokerBroker*) sur le réseau.

Nous voulons que l'AM soit un environnement spécialisable. Or le protocole de communication peut dépendre du domaine métier, il est donc important de bien modulariser chaque aspect du protocole de communication afin de pouvoir le substituer. Cependant, le protocole de communication est nativement déployé sur les AM d'un même domaine.

Architecture

Il faut distinguer deux éléments dans la communication : l'adressage et la technologie. Dans l'AM on distingue quatre éléments :

- L'objet du service : c'est le ou les interfaces d'un objet Java que l'on désire distribuer.
- Le servant : le servant abstrait l'objet du service pour la communication (similaire à de la réflexion Java). Il sert à s'interfacer avec un protocole de communication donné.
- Le proxy de communication : cet élément sert à communiquer avec un ou plusieurs objets distants. Le proxy de communication est appareillé avec l'implémentation du servant. Il masque la technologie utilisée pour la communication.
- Le proxy d'adressage : cet élément est la représentation distante d'un objet de service ; il implémente les mêmes interfaces que l'objet distant. Cet élément possède les informations nécessaires à la communication.

Dans notre prototype, la communication est basée sur le protocole HTTP. La fabrique de servant enregistre pour chaque objet de service une servlet dans le serveur HTTP déployé sur la plate-forme OSGi. Le servant est passé à l'instanciation à la servlet, qui transforme et redirige les requêtes HTTP en un appel Java sur l'objet du service. Le proxy de communication est un objet unique sur toutes les AM et partagé par tous les proxies d'adressage. Il se connecte à l'URL passée en paramètre, formate le message, envoie une requête GET ou POST (si la méthode invoquée renvoie un objet), puis coupe la connexion. Et finalement le proxy d'adressage est un proxy dynamique Java qui possède l'ensemble des

adresses réseau de l'AM distante, le port qu'utilise le serveur http de l'AM distante et l'identifiant de l'objet du service (cible la servlet).

Prenons l'exemple du diagramme de séquence de la Figure 80, l'objet *ClientHW* souhaite appeler la méthode *sayHello* de l'objet de service *HelloWorld* qui se trouve sur l'AM2. Pour cela, il appelle l'objet *ProxyGénériqueHW* qui implémente l'interface *HelloWorld*. Cet objet est le proxy d'adressage pour *HelloWorld*. Il connaît le ou les adresses de l'AM 2, le port qu'utilise l'AM 2 ainsi que l'identifiant de l'objet du service. Il appelle donc le proxy de communication qui formate le message et sérialise les paramètres de la méthode et effectue une requête HTTP en mode GET sur l'url : « http://192.168.0.2:8080/HelloWorld ». Cette requête est récupérée sur la servlet correspondante qui dé-sérialise et transmet le message au servant. Finalement le servant appelle la méthode *sayHello* sur l'objet *HelloWorld*. Le paramètre de retour est transmis sur le retour du GET.

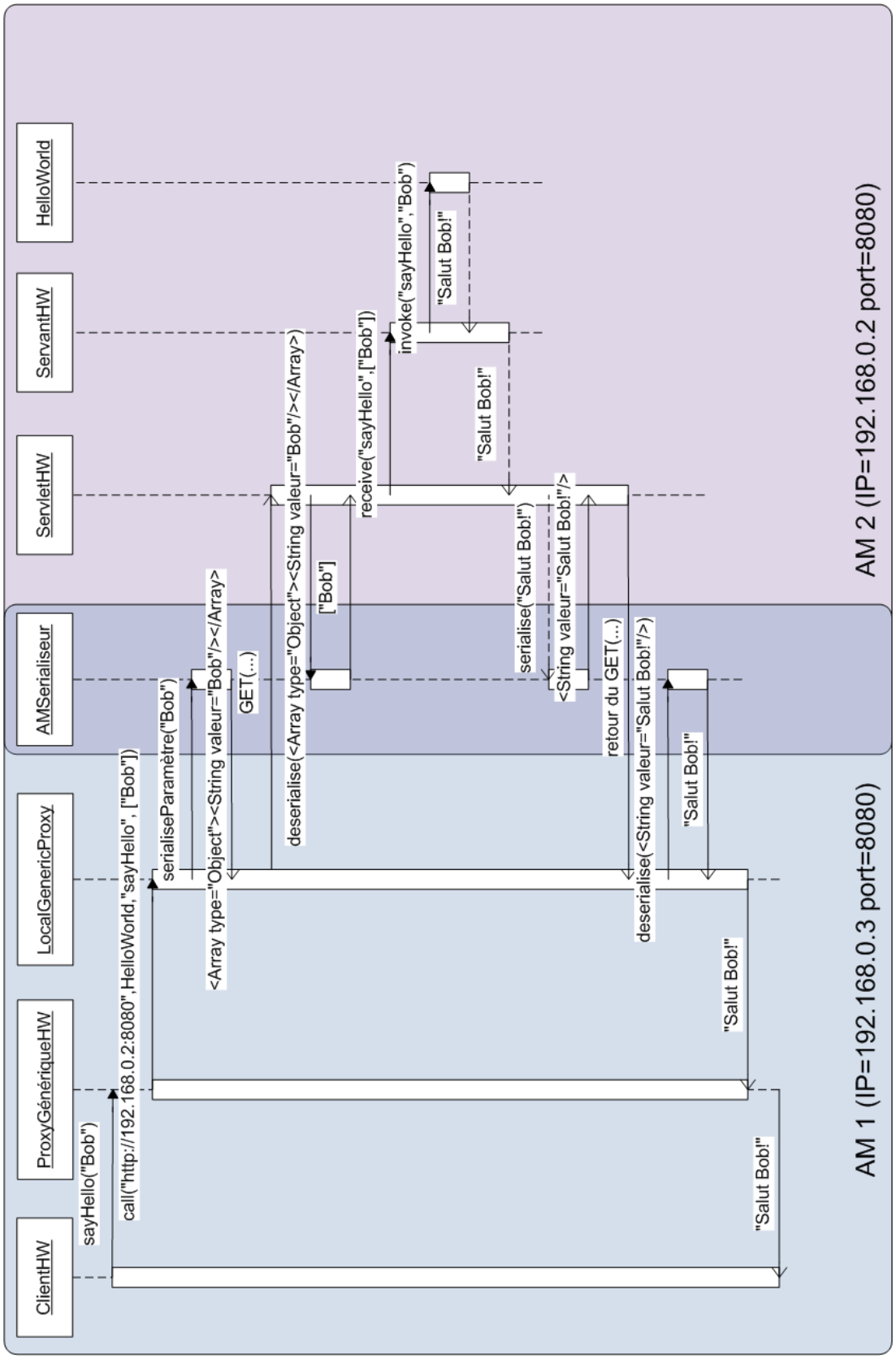


Figure 80 Diagramme de séquence d'un appel distant sur l'objet du service HelloWorld

Modularisation

Le diagramme de séquence de l'exemple précédent montre certains aspects intéressants :

- L'AM 1 et l'AM 2 partagent le même module de sérialisation, il peut être substitué.
- Le proxy de communication et le servant (servlet incluse) sont appareillés, nous pouvons changer le formatage (SOAP,…) ou la façon d'invoquer (REST, Web-Service,…) indépendamment du proxy d'adressage.

En d'autres termes, il est possible de substituer le protocole de communication, et donc de le spécialiser en fonction du domaine métier.

Multi-adresse

Un point important de notre approche est le fait qu'une AM peut avoir plusieurs interfaces réseau et donc plusieurs adresses pour la communication. Les proxies d'adressage connaissent ces adresses. Par conséquent si pour une raison quelconque l'objet du service venait à ne plus être accessible via une adresse (par exemple : délai expiré (*TimeOut*), *non atteignable (Unreachable*)) alors les autres adresses seront testées jusqu'à ce que l'une permette d'accéder à l'objet. Cependant, si aucune adresse ne permet d'y accéder alors une exception est soulevée.

Le protocole de communication a été testé selon le scénario suivant : les AM A, B et C sont connectées en wifi sur le réseau 192.168.0.0. Les AM A et C sont aussi connectées au réseau 10.0.0.0. Le test consiste à désactiver la carte wifi de l'AM A. Résultat : les proxies sur l'AM B soulèvent des exceptions alors que les proxies de A sur l'AM C utilisent l'IP 10.0.2.4. Inversement les proxies de B sur l'AM A soulèvent des exceptions alors que les proxies de C utilisent l'IP 10.0.2.41.

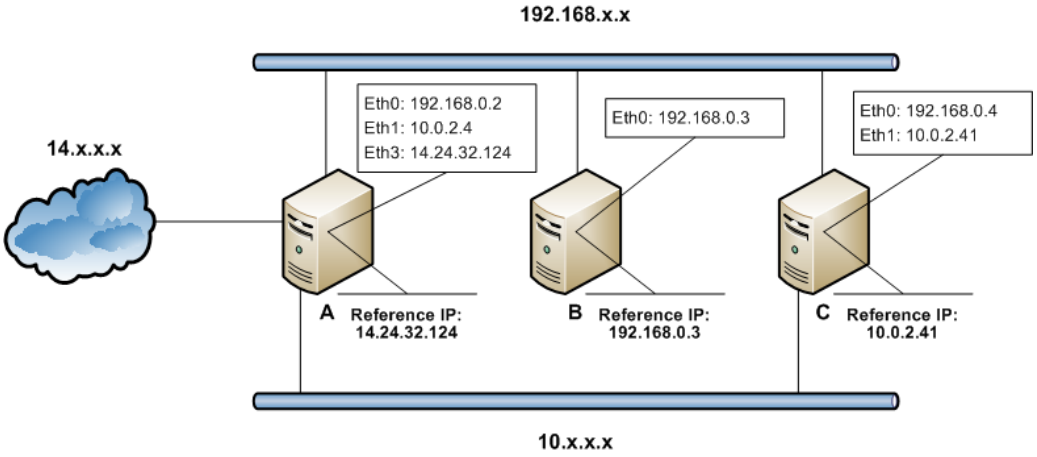


Figure 81 Topologie de l'exemple de test

2.1.2 Découverte

Le module de l'AM qui implémente la gestion et le registre de machine s'appelle *MachineBroker* ; la découverte est, quant à elle, déléguée à des modules externes. Nous avons donc développé un protocole de découverte dynamique basé sur l'envoi de paquet UDP sur une adresse multicast. Cette implémentation se nomme *AM-DynamicDisocvery*. Celle-ci se base sur l'envoi et l'écoute de quatre types de message (exemples à l'Annexe F -) :

- *Hello* : ce message notifie le démarrage d'une AM sur le réseau. Ce message contient le nom, l'identifiant, l'ensemble des adresses et les propriétés de l'AM. Il contient aussi la date et la durée de validité de la notification.

- *Bye* : ce message notifie le départ d'une AM sur le réseau. . Ce message contient le nom, l'identifiant et l'ensemble des adresses de l'AM.
- *isAlive* : ce message notifie la présence sur le réseau d'une AM. Il contient exactement les mêmes informations qu'un message *Hello*.
- *Probe* : ce message est utilisé pour la recherche active. Lorsque l'*AM-DynamicDiscovery* reçoit ce message, il renvoie aussitôt un message *Hello*.

L'*AM-DynamicDisocvery* est constitué de deux modules : un module d'envoi de messages multicast et un module d'écoute de messages multicast. L'écoute et l'envoi des messages se font sur **toutes les interfaces réseau** supportant le multicast. Le module d'envoi fonctionne de la manière suivante :

1. L'*AM-DynamicDiscovery* est déployé et activé sur la même plate-forme OSGi que l'AM.
2. Celui-ci découvre l'AM, récupère ses informations et envoie un message *Hello* sur le réseau.
3. L'*AM-DynamicDiscovery* envoie un message *Probe* pour forcer les autres *AM-DynamicDiscovery* à se déclarer afin d'accélérer la découverte.
4. Il envoie périodiquement – c'est-à-dire avant la fin du délai de validité – un message *IsAlive*.
5. Lorsque l'*AM-DynamicDiscovery* est arrêté, il envoie un message *Bye*.

Le module d'écoute fonctionne de la manière suivante :

- S'il reçoit un message *Hello* ou un message *isAlive* alors :
    1. Il le notifie au *MachineBroker* de l'AM locale.
    2. (*MachineBroker*) Si la machine n'était pas déjà découverte alors celle-ci est créée.
    3. Un minuteur est lancé en fonction du temps de validité du message.

    Ou

    2'. (*MachineBroker*) Si cette dernière était déjà découverte, alors les informations du message sont comparées aux informations du proxy de l'AM distante. Si les informations (par exemple la liste des adresses) sont différentes alors les informations sont mises à jour et une notification de modification est envoyée localement.

    3'. Le minuteur est mis à jour en fonction du temps de validité du message.
- S'il reçoit un message *Bye* alors il le notifie au *MachineBroker* de l'AM locale, et se charge de détruire la représentation de l'AM distante si besoin est.

Si un minuteur arrive à la fin de son compte-à-rebours alors, l'AM est considérée comme n'étant plus atteignable et la *MachineBroker* de l'AM locale est notifiée de son départ.

2.2 SPECIALISATION

Dans l'AM les extensions sont appelées *Broker*. Le gestionnaire et le registre d'extension sont implémentés par le module *BrokerBroker*. L'implémentation se base sur le registre d'OSGi, qui recherche tous les objets de service qui implémentent l'interface *fr.imag.adele.am.broker.Broker.java*.

Toutes les extensions peuvent être exposées. Dans le cas où les fonctionnalités du broker nécessitent un traitement particulier pour la distribution, il est possible de développer un proxy et un servant spécifique. Ils seront automatiquement chargés s'ils se trouvent dans le même package que l'implémentation du broker et que leur nom de classe soit la concaténation du mot *Proxy* ou *Servant* au nom de la classe du broker. C'est le cas des extensions *MachineBroker* et *BrokerBroker* de l'AM (cf. Figure 82).

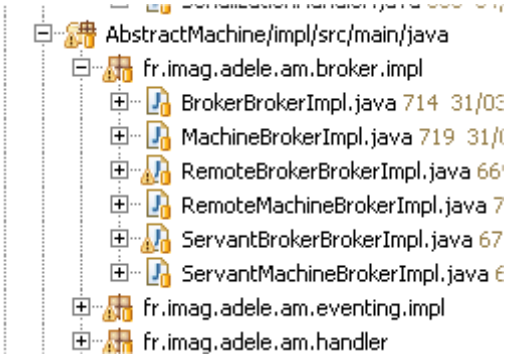


Figure 82 Exemples de proxies et de servants spécifiques

2.3 NOTIFICATION

Le mécanisme de notification de l'AM est implémenté par le module nommé *EventingEngine*. Il est basé sur le patron « publieur/souscripteur ». Sa particularité est d'être distribué et décentralisé ; c'est-à-dire qu'il permet de souscrire à des évènements venant d'autres AM sans avoir de serveur de message centralisé. Pour cela l'*EventingEngine* de chaque AM maintient la liste des souscripteurs qui le concerne. Si une AM est déconnectée d'une autre AM alors toutes les souscriptions qui viennent de cette dernière sont supprimées. Notre implémentation réutilise le protocole de communication de l'AM décrit ci-dessus.

Le démonstrateur développé pour le projet SEMbySEM (cf. section 1.1 de ce chapitre) est basé en grande partie sur le mécanisme d'évènement de l'AM. Le scénario prévu pour la démonstration sert de test de validation.

2.4 SYNTHESE

L'AM n'est pas l'élément central de cette thèse. Cependant elle soulève un bon nombre de défis, aussi bien techniques que de recherche, qui ne sont pas résolus dans cette thèse. Le prototype permet cependant de valider le concept d'AM comme étant un environnement réparti spécialisable.

L'AM est développée au-dessus d'une plate-forme OSGi utilisant la technologie iPOJO. L'AM utilise donc le modèle de composant orienté service d'iPOJO. L'implémentation du prototype a suffisamment découplé le protocole de communication, du protocole de découverte et du mécanisme d'évènement pour pouvoir les substituer en fonction du contexte du domaine métier. Il serait cependant intéressant de faire converger leurs implémentations vers des standards.

La Table 11 montre un certain nombre de métriques sur les cinq unités de déploiement (Jar) qui constituent l'AM. Ces résultats sont obtenus à l'aide de l'outil SONAR[21] sur la version 1.3.0 de l'AM. Les règles utilisées pour déterminer la compatibilité sont définies dans l'Annexe E -

[21] SONAR Codehaus. http://www.sonarsource.org/

Table 11 Quelques métriques sur l'AM

	Requis/Optionnel	Nb de classes	Nb de méthodes	Nb de lignes de code (JavaNCSS)	Nb de lignes de Javadoc	Rapport code / javadoc	Duplication	Compatibilité aux règles définies par sonar
API	R	28	90	462	492	51,6%	0,0%	72,9%
Implémentation	R	45	317	4751	1493	23,9%	3,6%	83,4%
Instance	R	2	5	71	21	22,8%	0,0%	78,9%
Module de Découverte	O	15	49	992	331	25,0%	15,0%	79,4%
Shell (Felix)	O	4	25	332	65	16,4%	0,0%	84,0%
Total		**94**	**486**	**6608**	**2402**	**26,7%**	**4,7%**	

3. SERVICES DYNAMIQUES ET HETEROGENES

Dans notre approche, la dynamicité et l'hétérogénéité des services sont adressées par SAM-CORE-RT. SAM-CORE-RT est un *runtime* d'intégration et est donc composé d'une partie abstraite et d'un ensemble de SCM.

La partie abstraite est décomposée en quatre modules :

- L'API de SAM-CORE-RT, qui contient le méta-modèle de SAM-CORE ainsi que les interfaces d'administration enregistrées dans le registre d'extensions de l'AM.
- L'API des SCM, qui contient les interfaces d'interaction avec les SCM propres à notre implémentation.
- Le module implémentation, qui contient l'implémentation de SAM-CORE-RT.
- Le module instance, qui contient les déclarations d'instances des différents composants de SAM-CORE-RT – dans notre cas un fichier de métadonnées iPOJO.

La Table 12 référence quelques métriques sur ces quatre modules à l'aide de l'outil SONAR[22] sur la version 1.3.0 de SAM-CORE-RT. Les règles utilisées pour déterminer la compatibilité sont définies dans l'Annexe E -

Table 12 Quelques métriques sur SAM-CORE-RT (partie abstraite)

	Nb de classes	Nb de méthodes	Nb de lignes de code (JavaNCSS)	Nb de lignes de Javadoc	Rapport code / javadoc	Duplication	Compatibilité aux règles définies par sonar
SAM-API	20	130	442	788	64,1%	0,0%	65,2%
SCM-API	3	47	123	329	72,8%	0,0%	59,3%
Implémentation	20	342	3361	1274	27,5%	5.5%	87,3%
Instance	0	0	0	0	0	-	-
Total	**43**	**519**	**3926**	**2391**			

Dynamicité

La dynamicité est un aspect difficile à évaluer. Dans notre cas, nous nous basons sur le démonstrateur pour SEMbySEM (cf. section 1.1 de ce chapitre) qui met en évidence les différents

[22] SONAR Codehaus. http://www.sonarsource.org/

aspects dynamiques que sont la disponibilité dynamique des AM et la disponibilité dynamique des services.

Hétérogénéité : SCM iPOJO

Dans cette section, nous allons nous intéresser exclusivement à la SCM iPOJO. En effet, bien que nous ayons développé différentes SCM (UPnP, DPWS et WS) dans le cadre de projets, ces dernières ne prennent en charge que l'import – c'est-à-dire instancier des proxies – des services. L'aspect d'intégration est donc limité à l'utilisation des services, car la manipulation de leurs cycles de vie n'est pas possible.

La SCM iPOJO est la plus intéressante car iPOJO est une technologie composant orienté service, et l'écart sémantique entre le méta-modèle de SAM-CORE et celui d'iPOJO est faible. Bien qu'il soit possible d'intégrer d'autres plates-formes, nous nous sommes essentiellement concentrés sur la SCM-iPOJO car elle est la technologie la plus proche et la plus compatible pour SAM-CORE-RT. En d'autres termes, la SCM-iPOJO est la plus complète et nous permet la quasi-totalité des opérations d'administration et de supervision.

Comme pour l'AM et SAM-CORE-RT, la spécification, l'implémentation et l'instance de la SCM sont séparées dans différents modules. L'implémentation de la SCM iPOJO implémente l'API SCM et est contenue dans le module (*bundle*) SCM-iPOJO-Impl. Les déclarations d'instances (métadonnées iPOJO) sont contenues dans le module SCM-iPOJO-Inst. La Table 13 fournit quelques métriques sur ces modules.

Table 13 Quelques métriques sur la SCM iPOJO

	Nb de classes	Nb de méthodes	Nb de lignes de code (JavaNCSS)	Nb de lignes de Javadoc	Rapport code / javadoc	Duplication	Compatibilité aux règles définies par sonar
SCM-iPOJO-Impl	23	260	3638	1066	22,7%	8,0%	86.1%
SCM-iPOJO-Inst	0	0	0	0	0	-	-
Total	**23**	**260**	**3638**	**1066**	**22,7%**	**8,0%**	

3.1 TESTS DE PERFORMANCES

Les tests ont eu pour but de comparer le surcoût en terme d'empreinte mémoire et de vitesse engendré par l'intégration de iPOJO dans SAM.

3.1.1 Protocole et configuration des tests

Les tests ont consisté :

1. à la création d'un arbre binaire d'un service implémentant la méthode *call* suivant un nombre d'instances donné.

```
public int call() {
  int ret = 0;
  if (this.instance1 != null) {
    ret = this.instance1.call();
  }
  if (this.instance2 != null) {
    ret = ret + this.instance2.call();
  }
  ret++;
return ret;
}
```

L'algorithme de la création de l'arbre est le suivant : on calcule la profondeur nécessaire de l'arbre pour le nombre d'instance donné. Puis on construit l'arbre, par un parcours en profondeur jusqu'à la profondeur que l'on a calculé précédemment à l'aide de la fonction suivante

$$f(x) = \left\lceil \frac{\log_{10}(\frac{x+1}{2})}{\log_{10} 2} \right\rceil$$

où x est le nombre d'instances souhaité.

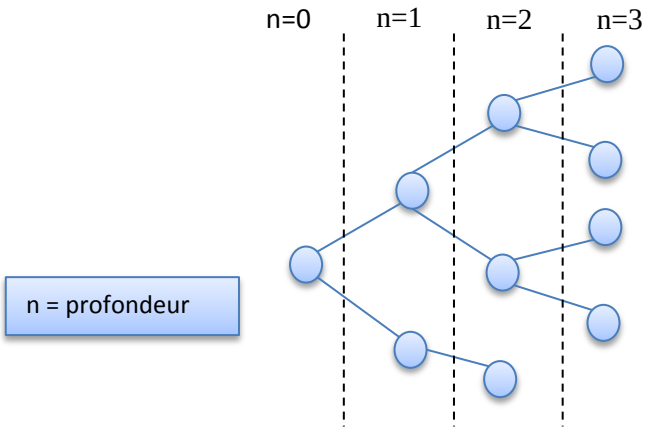
Soit en Java :

```
private int calculateI(double x) {
  double ret = 0;
  ret = Math.ceil (Math.log10((x + 1) * 0.5) / Math.log10(2));
  return Double.valueOf(ret).intValue();
}
```

Par exemple, si x=10 alors

$$f(10) = \left\lceil \frac{\log_{10}(\frac{10+1}{2})}{\log_{10} 2} \right\rceil = 3$$

Puisque l'arbre est construit en profondeur alors on obtient l'arbre d'instance suivant :



2. Relever l'empreinte mémoire à la fin de la création.

Avant la création de l'arbre le *garbage collector* est lancé puis l'empreinte mémoire est relevée. On fait de même après la création de l'arbre. La différence entre les deux nous donne l'espace mémoire utilisé pour un nombre d'instances donné.

3. Calculer le temps du premier appel à la méthode *call.*

Le temps est relevé avant et après l'appel à la méthode *call*.

4. Calculer le temps moyen pour 10 appels à la méthode *call*.

Le temps est relevé avant et après les 10 appels à la méthode *call* puis on divise le temps par le nombre d'appels.

Exploitation des résultats

Pour chaque nombre d'instance fixé, le test est relancé quatre fois. **Le meilleur et le pire résultat sont enlevés.** Le résultat final est la moyenne des deux autres.

Les algorithmes sont identiques seul l'instanciation diffère. Dans le cas de la SCM-iPOJO on a le code suivant :

```
…
Properties props = new Properties();
props.put(ComponentTestImpl.CURRENTLEVELPROPERTY, Integer.valueOf(i).toString());
this.instance1 = (ComponentTestImpl) ComponentTestImpl.implementation
                        .createInstance(props).getServiceObject();
…
```

Dans le cas d'iPOJO on a le code suivant :

```
…
Properties props = new Properties();
props.put(ComponentTestImpl.CURRENTLEVELPROPERTY, Integer.valueOf(i).toString());
ComponentInstance instanceRef = ComponentTestImpl.factory
         .createComponentInstance(props);
instanceRef.start();
this.instance1 = (ComponentTestImpl) ((InstanceManager) instanceRef)
        .getPojoObject();
…
```

Configuration du système

Les tests ont été lancés sur une machine ayant la configuration suivante :

Machine	OS	Windows XP (32bit) service Pack 3
	CPU	Intel Core 2 Duo E8600 : 3.33GHz par cœur.
	Bus	2.00GHz
	Mémoire (supporté)	3,25 Go de RAM
Contexte d'exécution Java	JVM en Mode Normal	JIT activé, mémoire max du tas 64 Mo
	OSGi	Version 2.0.4
	iPOJO	Version 1.6.0 (ipojo.proxy=disabled)

3.1.2 Analyses des résultats

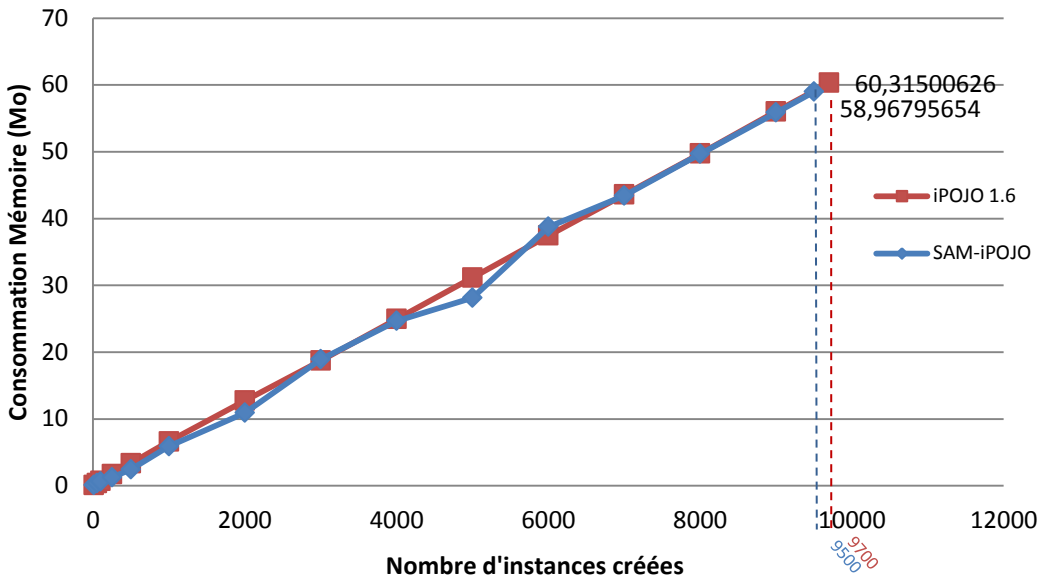


Figure 83 Consommation mémoire en fonction du nombre d'instances créées

Le graphique de la Figure 83 montre la consommation de la mémoire en fonction du nombre d'instances créées. Dans l'exemple suivant le surcoût est nul une fois les instances créées. Cependant les graphiques de la Figure 84 obtenue par la *visual VM* met en évidence que le surcoût est relatif au temps et non de la mémoire.

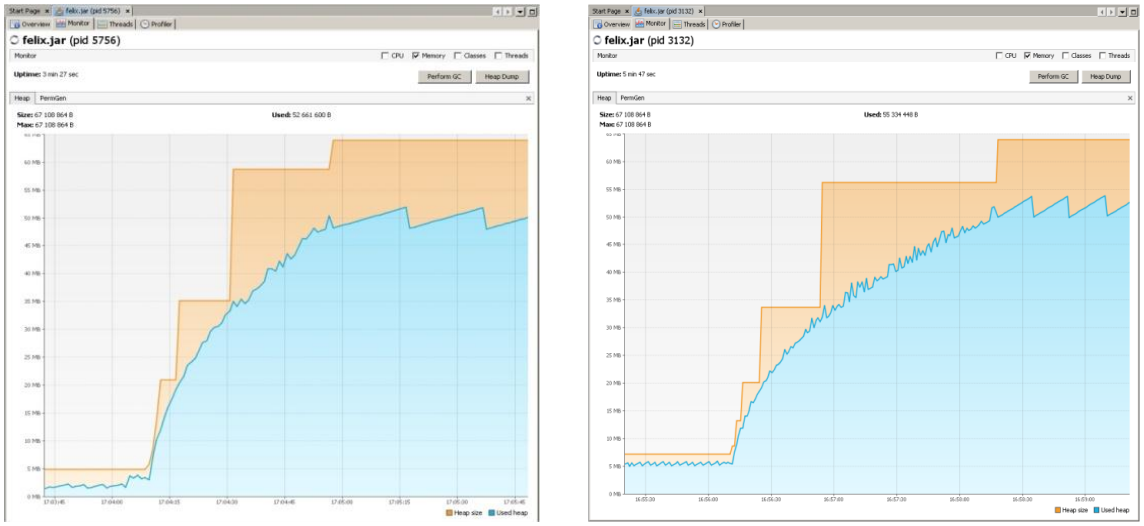


Figure 84 Courbe de consommation mémoire pour 4000 instances de iPOJO (à gauche) et de SAM (à droite)

Le graphique de la Figure 85 montre que le surcoût entre iPOJO et la SCM-iPOJO est faible. Noteons que l'inflexion de la courbe à partir de 500 instances est due à l'activation du JIT de la JVM. Outre l'anomalie pour 10 instances, le surcoût est en moyenne inférieur à 3% (précisément de 2,89787365%).

Figure 85 Temps moyen de 10 appels en fonction du nombre d'instances

En résumé, **l'impact en terme de mémoire et de temps d'appel dûs à l'intégration est négligeable**. Il faut cependant remarquer qu'il existe un coût en terme de temps d'instanciation dû aux calculs nécessaires à la transformation des opérations d'administration de SAM à iPOJO.

3.1.3 Comparaison avec d'autres technologies à service centralisées

Le protocole de test défini précédemment évalue l'empreinte mémoire et le temps d'invocation dans l'architecture de service dans SAM-RT. L'architecture est définie de manière programmatique dont les dépendances sont résolues à l'initialisation de l'application.

Pour situer notre intergiciel SAM-RT, nous allons porter le test précédent à trois environnements d'exécution pour des modèles d'application SCA : Tuscany 1.3, Tuscany 1.6 et Frascati 1.2 ; ainsi que sur l'environnement d'exécution Felix (implémentation d'OSGi version 4).

SCA spécifie un modèle de composant orienté service pour la définition d'application. Il ne spécifie pas comment et avec quelles technologies on doit implémenter un environnement d'exécution prenant en charge un tel modèle. Conséquemment SCA autorise – ou plus précisément n'interdit pas – d'implémenter un environnement d'exécution basé sur l'intégration d'autres technologies. De même SCA ne spécifie ni l'aspect dynamique ni la distribution ; et de ce fait laisse à la discrétion des développeurs de l'environnement de prendre en charge ou non ces aspects.

Nous ne nous comparons pas à l'approche SCA mais aux implémentations Tuscany 1.3, Tuscany 1.6 et Frascati 1.2 en tant qu'environnements d'exécution d'architecture orientée service. Il faut cependant noter que dans leurs cas, l'application de test est construite à partir d'un modèle par un gestionnaire d'application, alors que dans notre cas l'application est construite de façon programmatique par récurrence, comme pour le test implémenté au-dessus d'OSGi.

Le graphique de la Figure 86 montre la consommation mémoire utilisée en fonction du nombre d'instances créées.

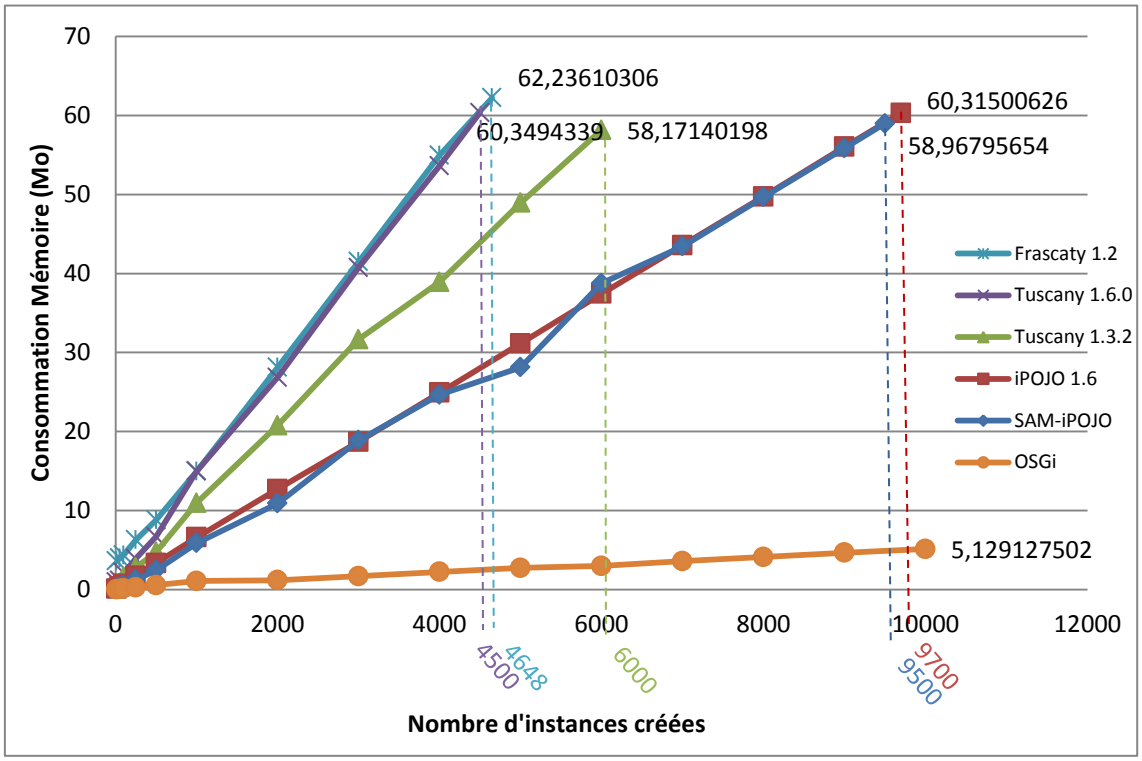


Figure 86 Consommation mémoire en fonction du nombre d'instances créées

Notre maîtrise de ces trois technologies n'est pas parfaite, un expert de ces technologies pourrait peut-etre obtenir de meilleurs résultats. Cependant l'écart est suffisamment important pour affirmer que l'espace mémoire nécessaire à une instance de composant de service prend nettement moins de place pour les technologies iPOJO et SAM-RT-iPOJO que pour les technologies SCA. Il faut toutefois remarquer que l'implémentation du même test pour l'implémentation OSGi sur Felix requiert 12 fois moins de mémoire pour 9500 instances.

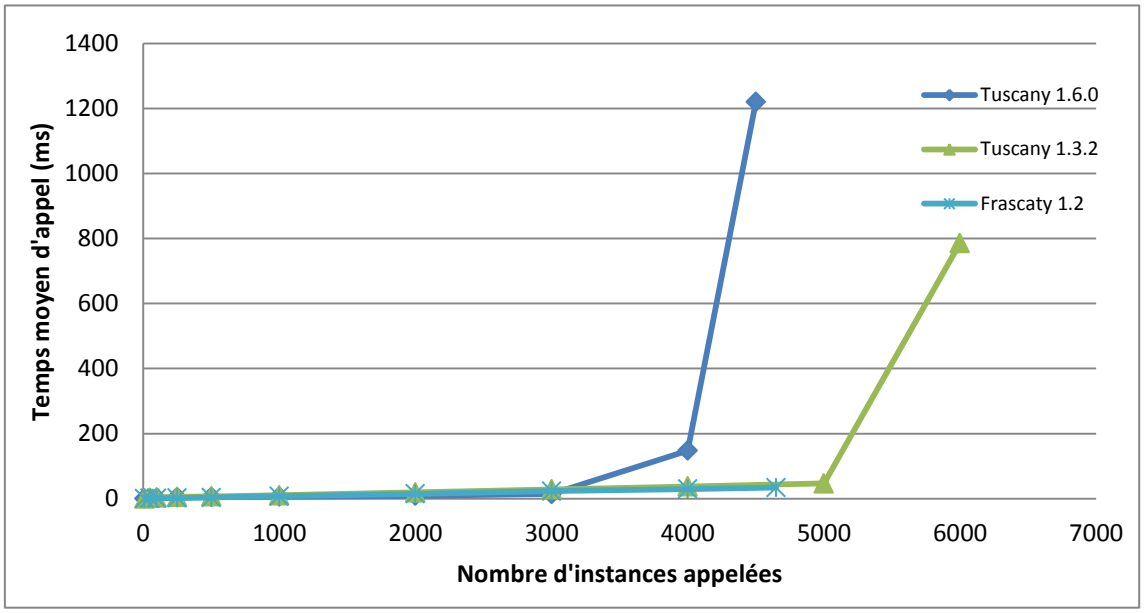


Figure 87 Temps moyen de 10 appels en fonction du nombre d'instances

Les graphiques de la Figure 87 et de la Figure 88 montrent le temps moyen nécessaire sur dix appels en fonction du nombre d'instances pour Tuscany 1.3, Tuscany 1.6, Frascati 1.2, SAM-RT-iPOJO, iPOJO 1.6.0 et Felix 2.0.4. Nous remarquons que dans le cas de SAM-RT-iPOJO et iPOJO, le temps moyen pour 9500 instances est dans le pire des cas inférieur à 0,2 ms ; alors que pour les technologies SCA, le temps moyen pour 10 instances est dans le meilleur des cas supérieur à 0,2 ms.

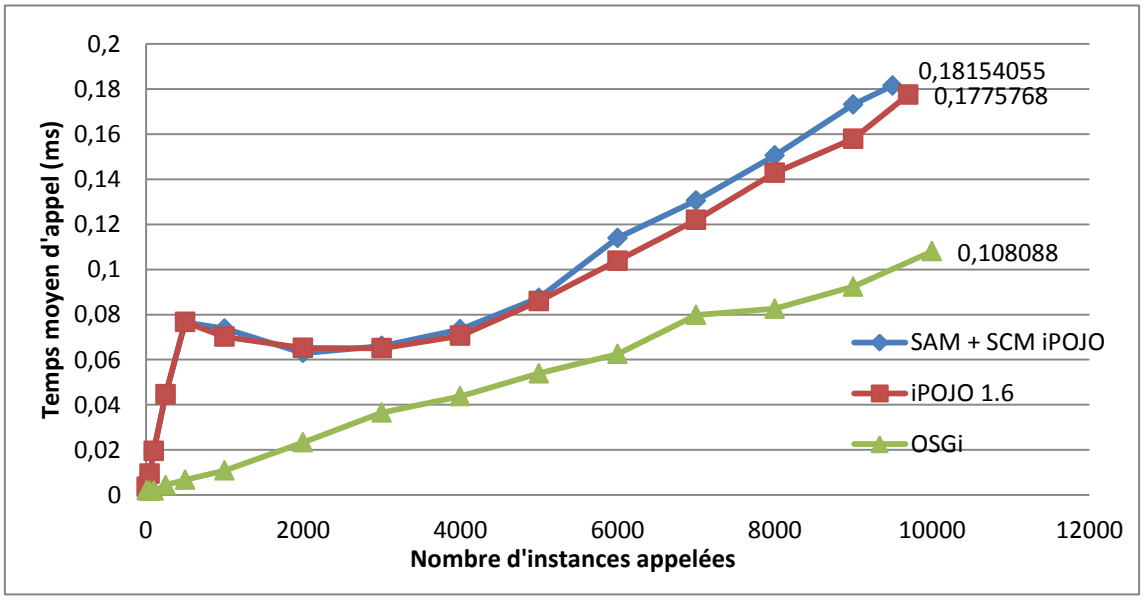


Figure 88 Temps moyen de 10 appels en fonction du nombre d'instances

Pour les tests sur OSGi, Felix a été lancé en mode serveur par conséquent le code est optimisé au démarrage et la courbe pour OSGi ne présente pas l'inflexion que l'on a pour SAM et iPOJO. A partir de 3000 instances, les résultats obtenus pour OSGi sont dans le pire des cas 43,8% inférieurs aux résultats pour SAM-RT et iPOJO.

3.2 SYNTHESE

L'expérience des différents projets et des différents tests réalisés nous a montré que le comportement dit dynamique d'une application est essentiellement caractérisé par la disponibilité de ses éléments. Le projet SEMbySEM montre qu'il est possible de définir une application qui se construit dynamiquement et évoluant en fonction des arrivées et départs d'objets administrables.

Dans le cas de l'hétérogénéité des services, il faut dissocier trois points :

- Intégrer un service dans SAM-CORE :
- intégrer les opérations d'administration du cycle de vie d'un service dans SAM-CORE-RT ;
- faire inter-opérer des services provenant de technologies hétérogènes via SAM-CORE-RT.

Le projet FOCAS utilise SAM-CORE-RT dans le premier but : il recherche des services distants et hétérogènes par le biais de SAM-CORE-RT dans le but d'associer un procédé à un service.

Le projet SELECTA, quant à lui, se concentre sur le deuxième point : le développement, l'installation et l'initialisation d'une application de services SAM.

Le troisième point est plus compliqué : SAM-RT étant développé au-dessus d'une plate-forme OSGi, il a été facile et rapide de développer des SCM pour les technologies UPnP, DPWS et Web-Service car ceux-ci se basent sur les ponts déjà existants pour OSGi. Conséquemment, les services issus de différentes technologies sont déjà dans le même espace d'exécution – celui d'OSGi – et déjà compatibles. Dans le cas des services distants qui ne sont pas dans SAM-RT*, ceux-ci sont des proxies n'ayant pas de dépendances : l'interopérabilité se résume donc à l'utilisation de proxies par un client dans le même espace d'exécution.

4. ETENDU A D'AUTRES PREOCCUPATIONS : UNITE DE DEPLOIEMENT ET DEPOT

Comme le définit la section 6 du Chapitre 3 - , en fonction des domaines métiers un certain nombre de préoccupations interviennent. Par exemple : le projet FOCAS/SAM (cf. section 1.2 de ce chapitre), requiert seulement SAM-CORE-RT ayant des SCM pour UPnP, DPWS ou simplement des *Web-Services*. Alors que dans le cas du démonstrateur pour SELECTA (cf. section 1.3 de ce chapitre), les préoccupations liées aux déploiements interviennent.

Nous allons traiter dans cette section l'aspect de l'extensibilité du méta-modèle de SAM-CORE par d'autres préoccupations. Notons cependant que l'implémentation est basée sur le mécanisme de spécialisation (registre d'extension) de l'AM. De plus, contrairement à l'exemple de la Figure 74, l'aspect ressource sera traité dans la même extension que l'unité de déploiement.

Le démonstrateur pour SELECTA introduit deux concepts : les unités de déploiement et les dépôts. Dans notre cas les unités de déploiement sont les conteneurs des concepts de SAM-CORE : Spécification, Implémentation et Instance. Indépendamment de la méthode de déploiement utilisée, les spécifications et les implémentations de services sont nécessairement contenues dans une unité de déploiement. Le concept de dépôt d'unité de déploiement dépend, quant à lui, de la méthode utilisée. Nous avons donc deux extensions du méta-modèle SAM-CORE : *DeploymentUnit-RT* et *Repository-SAM-RT*.

DeploymentUnit-RT

Nous avons défini le concept d'unité de déploiement de la façon suivante :

- Une unité de déploiement contient un ensemble de ressources identifiables qui – dans notre implémentation – sont des classes, des packages, des spécifications, des implémentations ou des déclarations d'instances.
- Une unité de déploiement est identifiée par son type (*bundle, Jar, contribution-SCA…*) et par un identificateur unique pour son type.
- Une unité de déploiement possède un ensemble de propriétés.

Les unités de déploiement sont accessibles via le registre d'unité de déploiement (*DeploymentBroker*) enregistré dans le registre d'extension de l'AM. Le registre d'unités de déploiement référence celles qui sont installées sur la plate-forme.

Soit le méta-modèle suivant :

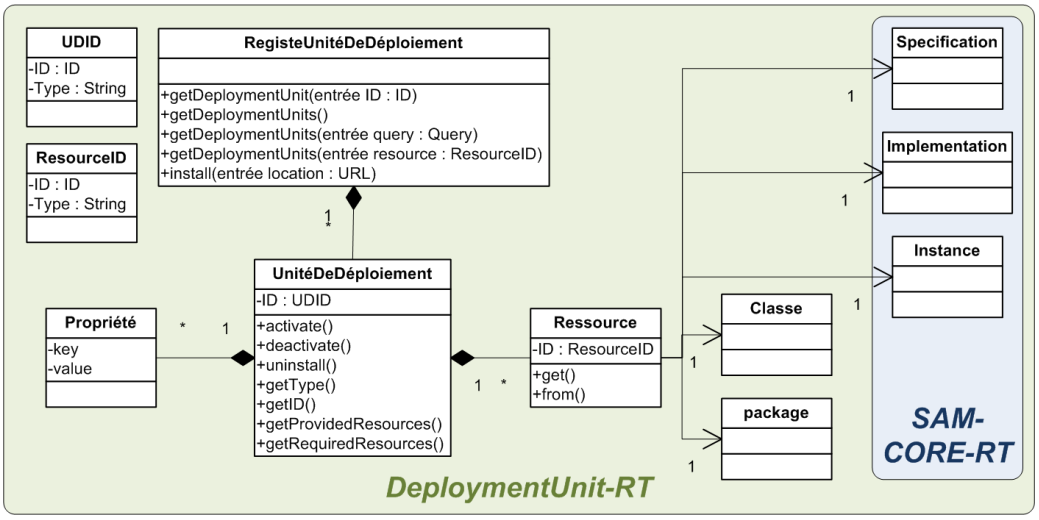


Figure 89 Un méta-modèle du concept d'unité de déploiement dans SAM

Le concept de ressource de la Figure 89 est une indirection vers la vraie ressource accessible via la méthode *get()*. L'identificateur de la ressource est unique dans l'unité de déploiement mais pas nécessairement sur la plate-forme.

Il est intéressant de noter deux choses : la première est qu'il est possible de tracer le graphe de dépendance en terme d'unité de déploiement permettant d'ordonnancer leur chargement et leur déchargement. La deuxième remarque est justement qu'une unité de déploiement n'est pas un élément statique mais qui possède **des états**. Dans notre cas une unité de déploiement possède trois états :

- Installé (*installed*) : c'est-à-dire transféré sur la plate-forme.
- Chargé (*activated*) : c'est-à-dire que les éléments de l'unité de déploiement sont chargés dans la plate-forme.
- Non-installé (*uninstalled*) : c'est-à-dire que l'unité de déploiement n'est pas présente sur la plate-forme.

Pour des raisons pratiques nous avons sommairement réutilisé le cycle de vie des *bundles* dans OSGi (cf. Figure 69). En effet nous ne cherchons pas à résoudre ce qu'est une unité de déploiement, mais définir une extension de SAM résolvant simplement un problème donné dans un démonstrateur.

Cependant le méta-modèle suivant montre un aspect très intéressant qui est l'extension des états d'un concept d'un autre *runtime.* En effet, dans SAM-CORE-RT, nous trouvons dans les registres

seulement les matérialisations des services chargés. Or dans le cas de *DeploymentUnit-RT*, il peut y avoir des matérialisations déclarées mais non chargées.

Dans l'exemple de la Figure 90, deux *bundles* iPOJO sont présents sur la plate-forme :

- Le bundle XYZ : qui contient trois composants (IA, IC et ID) ayant chacun une déclaration d'instance (A, C et D). Le *bundle* est activé, par conséquent les fabriques d'instance (IA, IC et ID) et les instances (A, C et D) sont donc en exécution.
- La bundle A : qui contient deux composants (IE et IG) et une déclaration d'instance E du composant IE. Ce *bundle* est installé mais non activé.

Pour des raisons de lisibilité nous omettons les dépendances de ressources de classes et de packages.

Le modèle de l'état d'exécution des services fourni par SAM-CORE-RT est donc composé des instances A C et D ainsi que des fabriques IA, IC et ID fournies par la SCM iPOJO. L'unité de déploiement (UD) de XYZ référence donc ces instances de concept. De plus, l'*Integration Machine* (IM) (cf. Chapitre 3 - 6.3) ajoute la propriété *state=activated* à ces concepts.

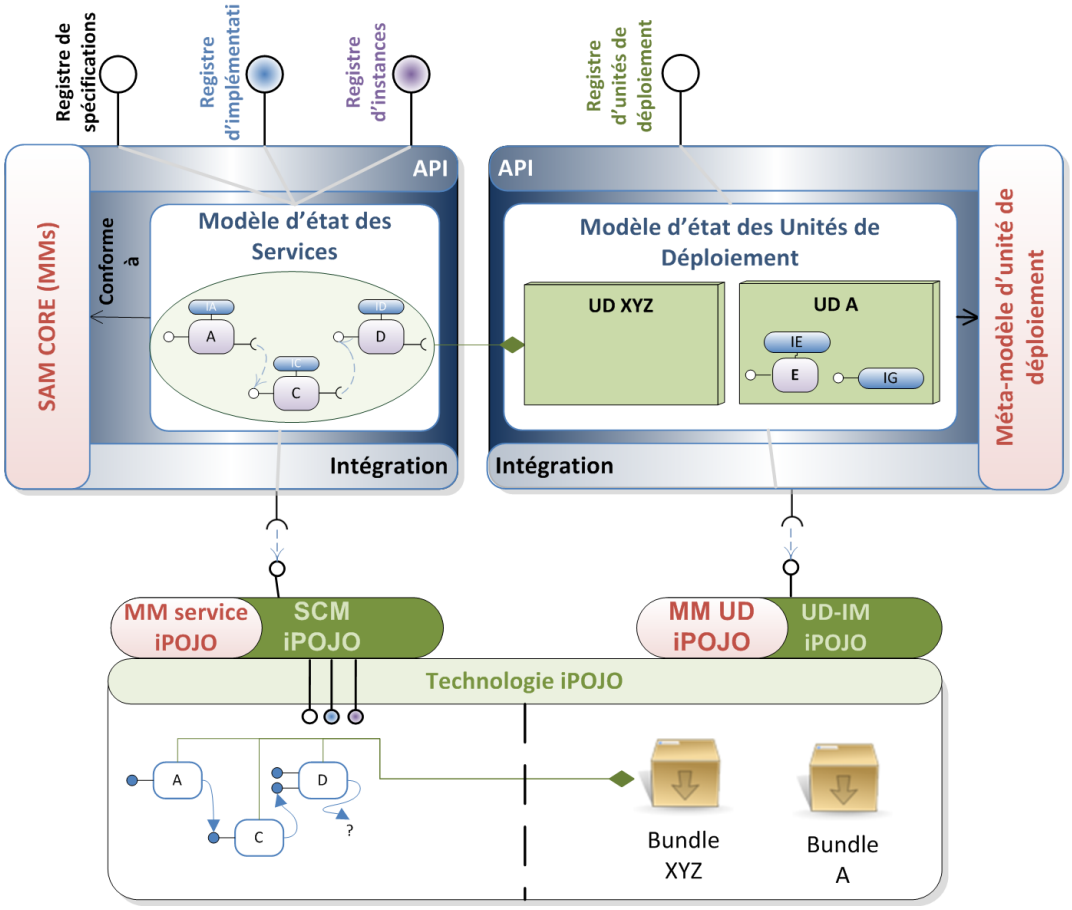


Figure 90 Exemple DeploymentUnit-RT

L'unité de déploiement A (UD A) fournit cinq ressources : l'instance E, les implémentations IE et IG ainsi que leur spécification respective. Cependant UD A n'est pas activée et par conséquent ses ressources ne sont ni chargées ni accessibles via les registres de SAM-CORE-RT. L'instanciation de leur représentation se fait donc dans le modèle d'état des unités de déploiement et celles-ci ont la propriété *state=installed*.

Repository-SAM-RT

Nous retrouvons aussi cette réutilisation de concepts dans l'extension *Repository-SAM-RT*. Cette extension a pour but d'intégrer le concept de dépôt d'unité de déploiement. Elle étend *DeploymentUnit-RT*.

Cette extension fonctionne de la manière suivante : une SAM peut être connectée à zéro ou plusieurs dépôts via des clients de dépôt. Ces clients sont accessibles via le registre de dépôt enregistré dans le registre d'extension de l'AM. Un client peut récupérer des unités de déploiement :

- Par leur identificateur ;
- Par une requête sur leurs propriétés ;
- En fonction d'une ressource donnée.

Il est possible d'installer une unité de déploiement à partir du dépôt. Soit le méta-modèle de la Figure 91.

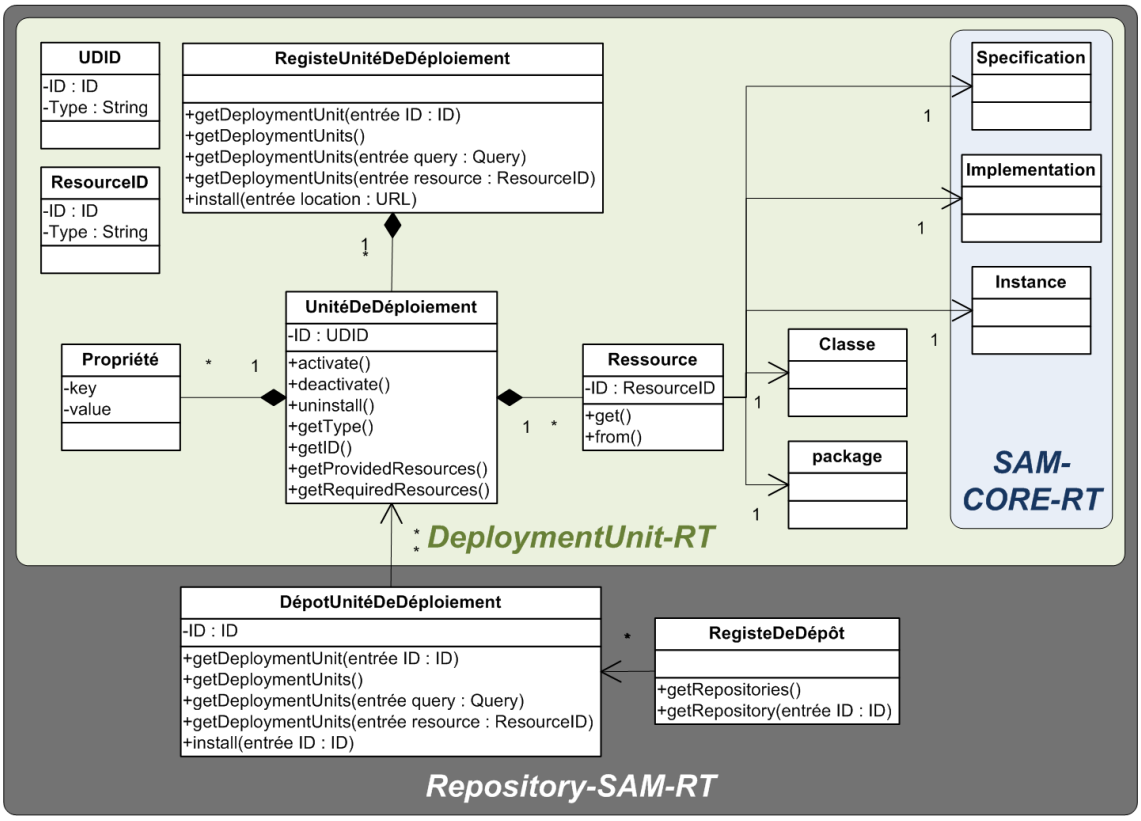


Figure 91 Un méta-modèle du concept de dépôt dans SAM

Nous nous retrouvons exactement dans le même cas de figure que dans l'extension de *DeploymentUnit-RT*, exceptés les points suivants : cette extension n'est pas une machine d'*intégration* (IM) : elle possède un espace d'exécution propre. *Repository-SAM-RT* étend *DeploymentUnit-RT* qui étend elle-même *SAM-CORE-RT.*

La Figure 92 reprend l'exemple de la Figure 90 en ne se focalisant que sur les modèles d'état. Le dépôt 1 contient les unités de déploiement XYZ et A. Ces unités sont déjà présentes sur la plate-forme d'exécution, par conséquent le client du dépôt référencera les instances UD XYZ et UD A du modèle d'état des Unités de Déploiement. Cependant le dépôt 2 contient une unité de déploiement UD B qui n'est pas installée sur la plate-forme. Les instances des concepts : Spécification, Implémentation et

Instance contenues dans UD B ainsi qu'UD B elle-même sont instanciées dans le modèle de l'état d'exécution des dépôts, ces instances ont la propriété *loadable=true*.

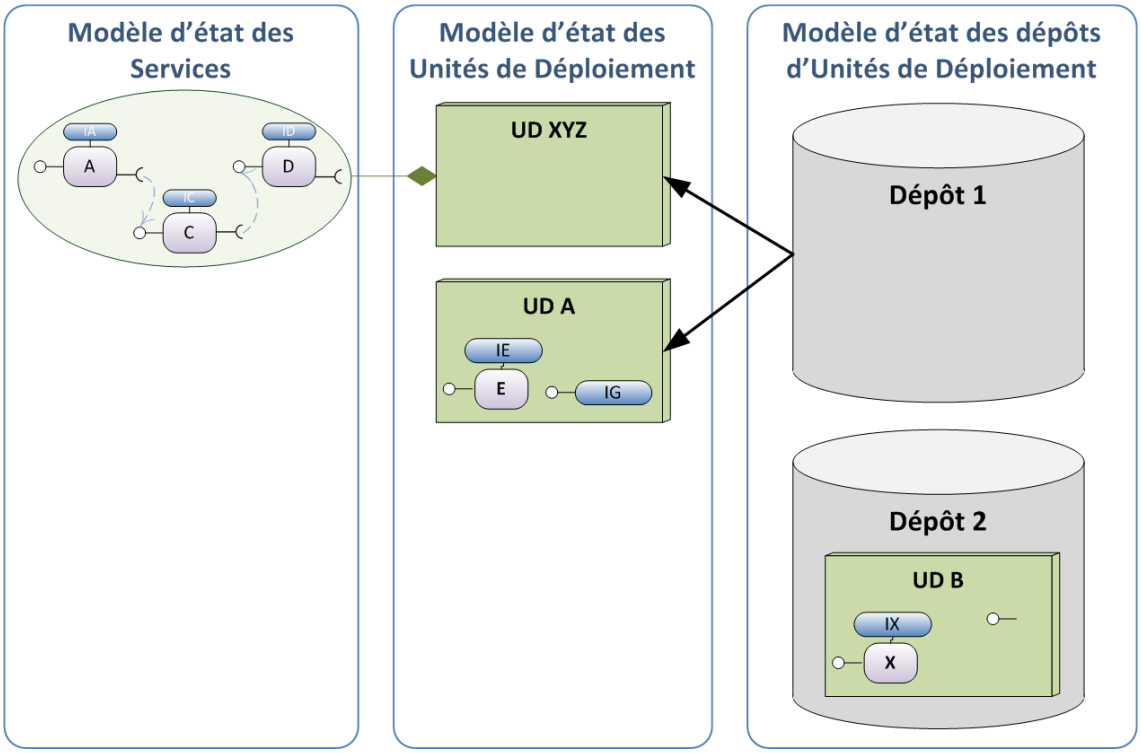


Figure 92 Exemple de modèles d'état

Synthèse

Nous voyons à travers ces deux exemples, que le méta-modèle de SAM-CORE peut être étendu par d'autres concepts et que ces concepts peuvent être enrichis par d'autres préoccupations.

Par exemple, SAM CORE ne définissait que les états *available* et *unavailable* pour les matérialisations d'un service. Cependant les deux extensions ont permis de raffiner ces états avec les propriétés suivantes : *installed, activated* et *loadable*. La Table 14 résume les états supportés par chaque concept.

Table 14 Etats supportés par les concepts en fonction des extensions

	Spécification		Implémentation		Instance	
SAM-CORE-RT	**available**	**unavailable**	**available**	**unavailable**	**available**	**unavailable**
DeploymentUnit-RT	*activated*	*Installed*	*activated*	*Installed*	*activated*	*Installed*
Repository-SAM-RT		*loadable*		*loadable*		*loadable*

Il est relativement facile d'étendre le méta-modèle SAM-CORE à l'aide d'extensions. En effet, mis à part l'ajout de propriétés, le modèle d'état de service reste inchangé.

En effet lorsqu'on ajoute des états à un concept, alors les clients d'origine de ce concept ne savent pas comment l'utiliser. Par exemple, un client de service ne sait pas comment utiliser une déclaration d'instance d'un service installée mais non activée. Pour cela il faut connaître les états et la sémantique liée, et donc être client de l'extension correspondante.

Cette remarque est vraie pour toutes les extensions. Outre les propriétés, aucun aspect de SAM-CORE n'est changé : pas de nouveau concept, pas de changement de sémantique (surcharge d'une opération d'administration)… Par conséquent il est toujours possible (modulo l'unicité de la clé des propriétés) d'étendre le méta-modèle de SAM-CORE-RT par d'autres concepts.

Cependant, l'extensibilité des *runtimes* est soumise à une contrainte non négligeable : une extension ne peut pas redéfinir ou étendre les définitions des concepts des autres extensions. En effet, la sémantique des méthodes d'administration est constante quelques soit les extensions présentes. Quand un client utilise une opération d'administration, il le fait dans un but ; surcharger ou redéfinir cette opération correspondrait à changer le but et donc rompre le contrat implicite entre un client et un service. Cependant, ajouter des opérations d'administration ne pose pas de problème conceptuel car cette méthode ne change pas la sémantique des opérations déjà existantes : mais cela reste un défi technique. Notre but est de définir un environnement pouvant s'adapter aux besoins de l'application ; **cette adaptation se fait dynamiquement,** c'est-à-dire sans couper ni relancer l'environnement, ce qui signifie qu'on ne peut pas ajouter des opérations d'administration. Pour expliquer cela, prenons le cas suivant : un client référence une implémentation de service qui est installable, mais non installée à partir de l'extension de dépôt. Ce client décide alors d'installer l'unité de déploiement de cette implémentation : la référence du client de l'implémentation du service est toujours valable, son état passe de *loadable* à *install*. Le client active l'unité de déploiement : l'état de l'implémentation est donc *activated* et peut donc être utilisé pour générer des instances. Cet exemple montre une chose très importante qui est **la continuité de la sémantique du concept :** seul change son état. Cette propriété forte qui permet de garantir à un client la sémantique des opérations d'administration, a pour contrepartie qu'il n'est pas possible de modifier l'interface d'administration d'origine ; il est bien sur possible d'en définir de nouvelles; car un client ne peut pas avoir dans son *classloader* plusieurs définitions d'une même interface et il n'est pas possible de modifier de façon transparente vis-à-vis du client la référence de l'instance du concept.

En résumé l'architecture mise en place permet d'adapter dynamiquement l'environnement d'exécution en fonction des besoins de ou des applications sans nécessiter l'arrêt de l'environnement tout en garantissant la continuité de la sémantique des opérations.

5. INFORMATIONS SUPPLEMENTAIRES ET SYNTHESE

Le but de la thèse est de définir et développer un environnement d'exécution dirigé par les modèles qui permet la description et l'adaptation des architectures basées sur des services hétérogènes.

Ce chapitre a montré que le prototype résultant a été utilisé à la fois dans un projet Européen : SEMbySEM, et dans des démonstrateurs au sein de notre équipe : FOCAS et SELECTA. Ces trois projets ont des buts différents et donc une utilisation différente de SAM-RT.

- SEMbySEM utilise principalement l'environnement réparti : AM.
- FOCAS utilise SAM-RT pour abstraire dans un modèle communs des services distribués et hétérogènes.
- SELECTA utilise SAM-RT dans la lignée de l'approche globale c'est à dire l'utilisation du modèle d'état pour superviser et modifier l'architecture des services à l'exécution.

Les résultats obtenus dans ces projets comme dans les différents tests effectués permettent de dire que les choix aussi bien conceptuels que techniques ne nous mènent pas à une impasse, au contraire, ils sont au moins conformes à nos attentes.

La Figure 93 met en évidence un ensemble de métrique sur la qualité du code. Les règles qui fixent les critères sont définies dans l'Annexe E - avec le nombre de violation pour chaque module (AM, SAM, SCM). De même il faut remarquer qu'il est très difficile de faire des tests unitaires avec la technologie OSGi et iPOJO, nous avons donc effectué des tests d'intégration avec l'outil Junit4OSGi[23].

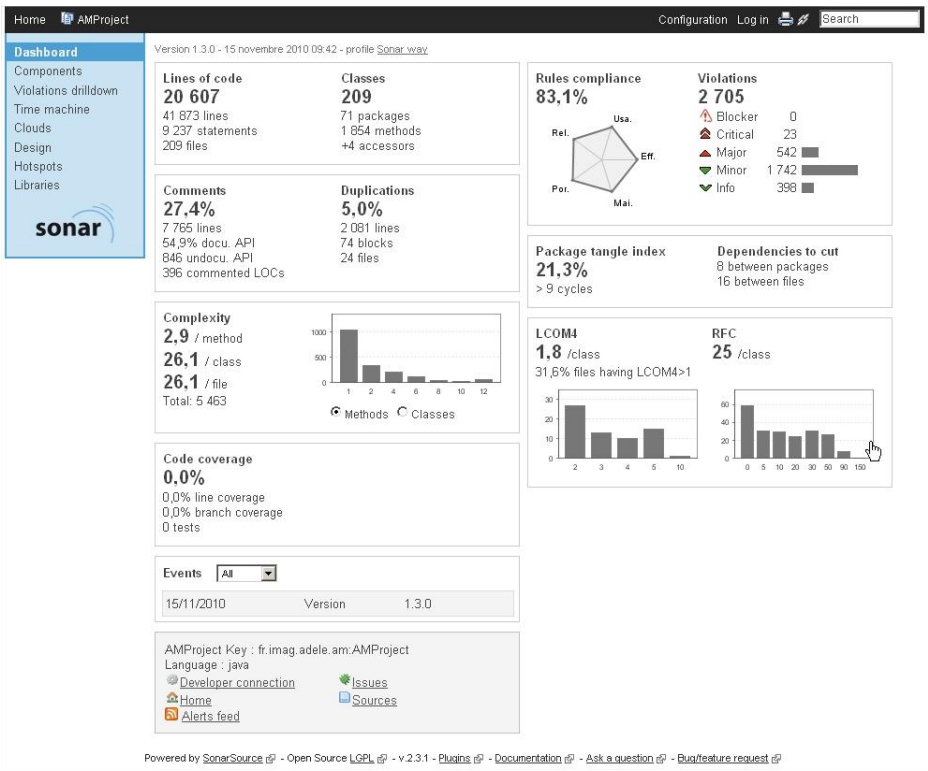


Figure 93 Quelques métriques sur l'ensemble du projet

[23] http://felix.apache.org/site/apache-felix-ipojo-junit4osgi.html

Chapitre 5 - Conclusion et Perspectives

L'approche orientée service (*Service-Oriented Computing* ou SOC) permet de composer des applications à partir d'éléments distribués et hétérogènes changeant, ainsi profondément le contexte d'exécution des applications du monde industriel. Or depuis peu, de nouveaux besoins émergent et guident le monde industriel vers le marché de l'informatique ubiquitaire (*ubiquitous computing*). L'informatique ubiquitaire consiste à intégrer dans une application des équipements informatiques (*device*) disséminés dans le monde réel.

L'interconnexion entre ces deux mondes constitue un réel défi pour l'exécution d'applications car une partie de leur architecture est statique alors que l'autre est dynamique, une partie des éléments de l'architecture sont administrables, et l'autre non … En d'autres termes, l'architecture de ces applications est constituée d'éléments ayant des comportements très divers et dont l'évolution semble non déterministe. Notre objectif est de contrôler l'évolution des architectures de telles applications.

Approche globale

Contrôler l'évolution des architectures de telles applications est un objectif à long terme. Pour atteindre cet objectif il est nécessaire d'aborder les aspects de conception, de déploiement et d'exécution d'application. Nous proposons dans notre approche de transférer une partie de la connaissance des éléments et de l'application issue de la phase de conception à la phase d'exécution, pour pouvoir diriger l'adaptation de l'architecture. Cette adaptation passe par la comparaison de l'état de l'architecture en cours (propriétés intrinsèques et contextuelles des éléments) avec la définition (*Connaissance*) de l'application.

Sujet de thèse

Cette thèse propose un environnement d'exécution dirigé par les modèles pour la description et l'adaptation d'architecture à service hétérogène. Le but est de fournir un modèle **descriptif** et **prescriptif** de l'état de l'architecture sur lequel on pourra baser l'analyse et la comparaison avec le modèle de l'application souhaitée. Ce modèle, dit model@runtime :

- Cache l'hétérogénéité des technologies orientées service ;
- Réagit à la disponibilité dynamique des éléments qui la composent ;
- Est générique et spécialisable pour pouvoir définir des mécanismes d'adaptation en fonction d'un domaine métier.

Pour cela l'environnement d'exécution doit définir :

- Un méta-modèle de service offrant les concepts nécessaires pour l'adaptation dynamique des applications ;
- Un mécanisme d'intégration des technologies SOA dans un modèle homogène conforme au méta-modèle défini précédemment ;
- Un mécanisme d'intégration de nouvelles préoccupations comme le déploiement ;
- Un moyen d'interaction entre le niveau « exécutif » (service) et le niveau « décisionnel » (application).

1. CONTRIBUTIONS

Cette thèse propose un environnement d'exécution – nommé SAM-RT – qui aborde cette problématique. Dans le cadre de cette thèse nous traitons l'architecture de cet environnement en quatre éléments fonctionnels :

- Un environnement réparti *Abstract Machine* (AM);
- Un méta-modèle descriptif et prescriptif d'architectures basées sur des services hétérogènes, fourni par SAM-CORE-RT;
- Une intégration des services dans un modèle homogène et leur interopérabilité par des *Service Concrete Machines* ;
- Une extensibilité du modèle de l'état d'exécution à d'autres préoccupations par des modules appelés *runtime*.

1.1 ENVIRONNEMENT REPARTI

Un des aspects sous-jacents à la problématique est le fait que les applications soient distribuées, mais que les environnements d'exécution soient, eux, généralement centralisés. Ce cloisonnement des environnements fait que la C*onnaissance* de l'application est éclatée ; la vision d'ensemble nécessaire à l'analyse et au raisonnement pour l'adaptation de l'application n'est pas possible. Nous avons donc distribué l'environnement d'exécution pour, d'une part, identifier la source d'un service, d'éviter les duplications et les boucles d'appels liées à l'éclatement de la connaissance de l'application ; et d'autre part pour pouvoir partager l'information de l'état d'exécution afin d'obtenir cette vision d'ensemble.

Il faut noter que le concept d'environnement réparti n'est pas propre à l'approche orientée service ; nous avons donc défini un environnement, nommé *Abstract* Machine (*AM*) qui regroupe les aspects de découverte, de communication et de notification. Cet environnement restreint sert de fondement à la conception d'environnements pour un domaine d'utilisation précis nécessitant la répartition spatiale.

Dans notre cas l'*Abstract Machine* a été spécialisée en environnement d'exécution d'architectures basées sur des services hétérogènes ; cet environnement spécialisé se nomme SAM-CORE-RT.

1.2 META-MODELE DESCRIPTIF ET PRESCRIPTIF D'ARCHITECTURES BASEES SUR DES SERVICES HETEROGENES

Notre environnement d'exécution est basé sur des services hétérogènes dans le but de les assembler et les faire inter-opérer dans une application. Nous avons abstrait leurs technologies pour obtenir un modèle homogène des services qui s'exécutent. Nous basons l'adaptation sur cette abstraction.

Ce modèle de l'état d'exécution des services est défini par le méta-modèle SAM CORE. En partant du constat que pour pouvoir adapter une application il faut d'une part connaître son état et d'autre part pouvoir agir sur son état, nous avons basé SAM-CORE sur une approche de composant orienté service. En effet, cette approche spécifie les concepts d'instance et d'implémentation.

- Avoir les concepts d'instance et de dépendance permet de définir une architecture, ce qui dans notre cas permet de décrire l'état d'exécution d'une plate-forme à service par un modèle **descriptif** de service.
- Avoir le concept d'implémentation permet d'agir sur le cycle de vie d'une instance de service car, à partir d'une implémentation, il est possible d'instancier, de supprimer, … En résumé avoir le concept d'implémentation permet de définir une partie des opérations de manipulation de

l'architecture. De ce point de vue le modèle d'état devient **prescriptif** car il permet la modification de l'architecture.

En outre SAM-RT fournit un registre pour chaque matérialisation de service : spécification, implémentation et instance ; ainsi qu'un mécanisme de notification pour signaler les changements d'état (apparition ou disparition) des différentes matérialisations. En résumé SAM-RT fournit un ensemble de registres permettant la sélection d'un service ainsi qu'un mécanisme de notification de la disponibilité dynamique des services. SAM-RT possède donc toutes les propriétés d'un **SOA dynamique**.

1.3 INTEGRATION ET INTEROPERABILITE DES SERVICES

Notre environnement d'exécution utilise des technologies à service hétérogènes. Non pas par défi technique, mais parce que chaque technologie aborde un problème précis : pour faire interopérer des systèmes, pour pouvoir communiquer avec des équipements domestiques, pour concevoir une plate-forme d'exécution à la carte… L'utilisation de technologies dépend avant tout du besoin du domaine métier. Or, les besoins actuels tendent à l'utilisation simultanée de différentes technologies. Cette multiplication des technologies à service dans un même domaine devient problématique car les technologies ne partagent pas leurs connaissances comme l'identification, la syntaxe ou la sémantique.

Au final, le défi ne vient plus de la résolution d'un problème par une application, mais de la complexité et de l'incompatibilité des technologies utilisées pour définir l'application. D'où la volonté d'intégrer des technologies SOA dans un modèle homogène abstrayant la technologie et de définir un mécanisme pour faire interopérer ces technologies.

L'environnement d'exécution SAM-CORE-RT se base sur des *Service Concret Machines* (SCM) qui se chargent d'une part : **d'intégrer** les services s'exécutant sur une technologie à service dans SAM-CORE-RT, et d'autre part d'intégrer les services de SAM-CORE-RT dans la technologie cible, dans le but de faire **interopérer** les services issus de technologies hétérogènes.

En résumé SAM-CORE-RT est un SOA qui a pour but d'unifier différentes technologies à service dans un même modèles. Cependant, SAM CORE est un méta-modèle composant orienté service sémantiquement plus riche que la plupart des approches orientées service. Par conséquent, en fonction de la technologie intégrée, il existe des écarts sémantiques. Toutefois, SAM-CORE-RT fournit au moins autant, sinon plus, d'informations que la plate-forme d'origine et peut donc se substituer à celle-ci.

Il faut bien noter que l'intégration dépend de l'implémentation des SCM. Une même technologie peut être intégrer différemment dans SAM en fonction des choix d'implémentation de la SCM.

1.4 EXTENSIBILITE DES PREOCCUPATIONS

Dans beaucoup de domaines, l'état de l'architecture n'est pas nécessairement suffisant pour prendre une décision d'adaptation. De plus, les opérations de création et de destruction d'instances ne sont pas suffisantes pour adapter l'architecture aux besoins de l'application. Les préoccupations comme le déploiement, la gestion des ressources ou la charge mémoire peuvent être requises.

Nous avons donc défini un mécanisme d'extensibilité du modèle d'état à des préoccupations autres que celles des services. Pour cela nous avons généralisé le mécanisme de SAM-CORE-RT sous le concept de *runtime*. Notre concept de *runtime* a pour but de prendre en charge un point de vue particulier de l'environnement d'exécution. Le point de vue est défini d'une part par un méta-modèle qui décrit les concepts associés et leurs propriétés, et d'autre part par un ensemble d'états pour chaque concept du méta-modèle. Celui-ci peut étendre un ou plusieurs autres *runtimes*, sur lesquels il peut ajouter des propriétés. Un point de vue peut dépendre des technologies sous-jacentes et dans ce cas le *runtime* est appelé *integration runtime*. Le *runtime* fournit une API correspondant à ce point de vue. Cette API **doit** fournir au moins les actions pour modifier les états définis dans le schéma de transition d'état.

Les *runtimes* ont à peu près le même niveau d'abstraction : leur but est d'étendre le méta-modèle SOA avec de nouveaux concepts, de nouvelles fonctionnalités ou de nouvelles propriétés. Ce mécanisme d'extensibilité peut être considéré comme une intégration horizontale, par opposition à l'intégration verticale qui consiste à intégrer des technologies existantes dans un modèle commun situé à un niveau supérieur d'abstraction.

De même que SAM-CORE-RT utilise les *SCM* pour intégrer les différentes matérialisations de service d'une technologie dans le modèle de l'état d'exécution des services, les *integration* runtimes utilisent les *Integration Machine* (IM) pour intégrer les différents éléments d'une technologie dans le modèle de l'état d'exécution.

En résumé, une IM est un médiateur entre un système (réel) et un *runtime* abstrait. L'IM intègre les éléments identifiés par le méta-modèle associé à son *runtime*, en assurant l'alignement sémantique et syntaxique entre le système et le modèle du *runtime*.

1.5 UTILISATION DANS DES PROJETS

Le prototype résultant de cette thèse a été utilisé principalement dans trois projets :

- Le projet Européen SEMbySEM (ITEA2) pour lequel on utilise principalement les fonctionnalités de l'environnement réparti ;
- Un des prototypes de FOCAS issu de la thèse [PedT09] utilise SAM-CORE-RT pour abstraire la technologie du service pour l'orchestrateur ;
- Le démonstrateur de SELECTA issu de la thèse de [DieT10] utilise un manageur d'application basé sur les modèles de l'état des services et des unités de déploiement de SAM-RT pour initialiser une application à partir d'un modèle par intention de l'application souhaitée.

Ces trois projets utilisent SAM-RT pour des objectifs différents, mais qui ont comme point commun l'utilisation de services pour la conception d'application : dans le premier cas l'application est définie de manière programmatique, dans le deuxième cas les services sont utilisés pour implémenter les fonctionnalités des activités d'un modèle d'orchestration, et dans le dernier cas un manageur d'application génère une architecture à base de services hétérogènes à partir d'un modèle d'application.

Les projets en cours tendent à résoudre l'étape suivante qui est la conception et le développement d'un manageur d'application initialisant et adaptant l'architecture de l'environnement d'exécution en fonction des aléas du contexte.

1.6 SYNTHESE

Ces différents éléments architecturaux permettent de produire un middleware réparti fournissant un modèle lié causalement à l'architecture en cours d'exécution. Le méta-modèle :

- est basé sur le paradigme composant orienté service qui permet de définir différents types d'applications : orchestration, composition structurelle à partir d'un ADL ou de manière programmatique…
- couvre les différents méta-modèles à service existants, permettant d'abstraire sans perte d'information les technologies à service existantes, cachant ainsi l'hétérogénéité technologique de l'architecture en cours d'exécution.

Le modèle est lié causalement au système : d'une part toute modification du modèle d'architecture engendrera une modification de l'architecture en cours d'exécution, et d'autre part toute modification de l'architecture en cours d'exécution en réaction par exemple à la disponibilité dynamique des éléments qui la composent engendrera une modification du modèle.

Finalement, le modèle réflexif de l'architecture peut être étendu à d'autres aspects à l'aide de modèles pouvant être juste descriptifs (cf. modèle de contexte Figure 22) ou réflexifs (liés causalement au système).

La Table 15 résume les différentes caractéristiques de l'environnement d'exécution SAM-RT.

Table 15 Caractéristiques de SAM-RT

Propriété	SAM-RT
Environnement	Réparti
Paradigme	Composant orienté service
Type d'application	Dépend du gestionnaire d'application
Système ciblé	Technologies orientées service
Répartition des éléments de l'application	Dépend des technologies intégrées
Type d'adaptabilité	Dépend du gestionnaire d'adaptation
Sondes et effecteurs	Modèle réflexif de l'architecture en terme de services pouvant être étendu à d'autres préoccupations
Opérations d'adaptation	Par défaut et en fonction de la technologie sous-jacente : création et destruction d'instance, modification des dépendances

2. PERSPECTIVE DE RECHERCHE

Les perspectives de recherche découlant de cette thèse se divisent en deux principaux axes :

- L'environnement réparti ;
- L'infrastructure de l'approche globale.

2.1 L'ENVIRONNEMENT REPARTI

Dans le Chapitre 3 - Contribution section 3, nous avons abordé un ensemble de problématiques liées à la distribution sans toutefois les traiter.

Itinérance

Actuellement, nous avons limité **l'itinérance** (*roaming* en anglais) au niveau de l'environnement d'exécution. C'est-à-dire que nous sommes capables d'identifier une *Abstract Machine* qui disparaît d'un réseau pour apparaître sur un autre, et d'affirmer ou d'infirmer s'il s'agit du même objet. Il serait intéressant d'étudier comment étendre l'itinérance aux objets des *Abstract Machines* ; c'est-à-dire d'être capable – dans le cas où un service disparaît d'une machine pour apparaître sur une autre – d'affirmer ou d'infirmer s'il s'agit du même service qui a migré ou bien s'il s'agit d'une coïncidence où un service de même type a été activé juste après la disparition de l'autre.

Multi-protocole et multi-chemin

Nous autorisons la coexistence de plusieurs protocoles de communication sur le même environnement d'exécution, et la connexion de cet environnement à plusieurs réseaux. Par conséquent pour interroger une machine distante, nous avons une multitude de protocoles de communication couplés à une multitude de chemins possibles. Dans le cas actuel, un client utilise soit son protocole de communication soit celui par défaut de l'environnement. Le chemin est, quant à lui, déterminé par la première adresse réseau trouvée valide. Il serait intéressant que le choix du protocole de communication ainsi que du chemin à prendre soit pris en charge par un mécanisme de la plate-forme qui les détermine en fonction du contexte.

Topologie : domaines et portées

Dans notre cas la découverte peut se faire de deux manières : soit via un protocole de découverte dynamique, soit par une connexion statique. Nous supposons que deux machines s'étant découvertes peuvent se connecter l'une à l'autre. Or cela n'est pas toujours vrai : par exemple pour des raisons de sécurité : pare-feu ; ou parce que l'on a découvert une machine d'une manière ou d'une autre dont le réseau n'est pas accessible à partir de la machine locale.

Les problèmes décrits ci-dessus ont déjà été abordés dans le cadre de l'infrastructure de l'internet – qui est un type d'environnement réparti homogène.

Il serait donc intéressant de concevoir l'environnement réparti comme un ensemble d'environnements hiérarchiques interconnectés. Cette question est liée aux concepts de portées et de domaines.

2.2 APPROCHE GLOBALE

Cette thèse a défini un environnement d'exécution dirigé par les modèles permettant la description et l'adaptation des architectures à services hétérogènes. Cet environnement, schématisé par la Figure

94, est la fondation sur laquelle les niveaux de composition d'application et d'administration peuvent être bâtis.

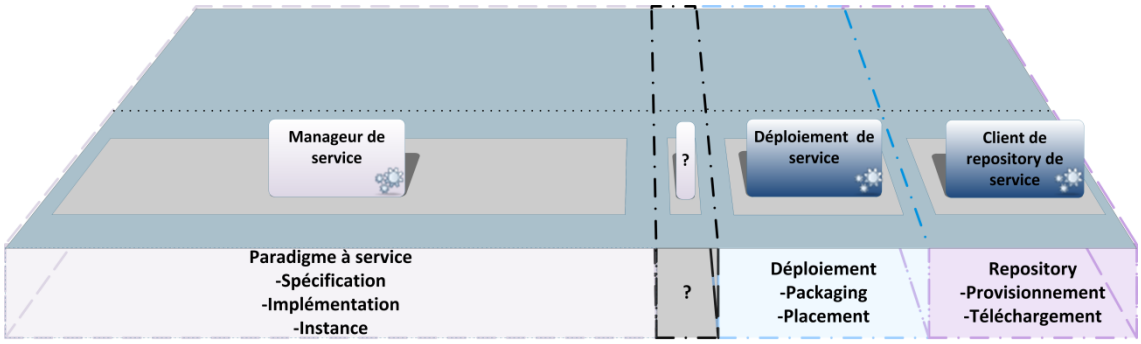


Figure 94 Réalisation de la thèse

Les trois projets : SEMbySEM, FOCAS et SELECTA, ont construit une application au-dessus de SAM-RT :

- Dans SEMbySEM, l'application est définie de manière programmatique et l'architecture évolue en fonction des aléas du contexte. Cependant, cette évolution est subie : on n'adapte pas l'architecture.
- Dans FOCAS, l'application est construite par orchestration. Il est possible de substituer un service par un autre ou d'adapter l'architecture des activités, mais en aucun cas FOCAS n'adapte l'architecture de l'environnement d'exécution.
- Dans le cas de SELECTA, l'application est définie à partir d'un modèle d'architecture par intention qui, à l'exécution, est interprété par un manageur d'application qui se charge de générer une architecture *valide*. Une fois l'architecture initialisée, celle-ci peut évoluer en fonction des aléas du contexte mais ne sera pas adaptée par le manageur d'application.

Ces trois projets n'initialisent ni adaptent l'architecture de l'environnement d'exécution en fonction des aléas du contexte. Pour cela il nous reste un ensemble de travaux à réaliser défini par notre approche globale.

La figure ci-dessous détermine les travaux en cours ou à explorer pour atteindre cet objectif (cf. : Figure 37).

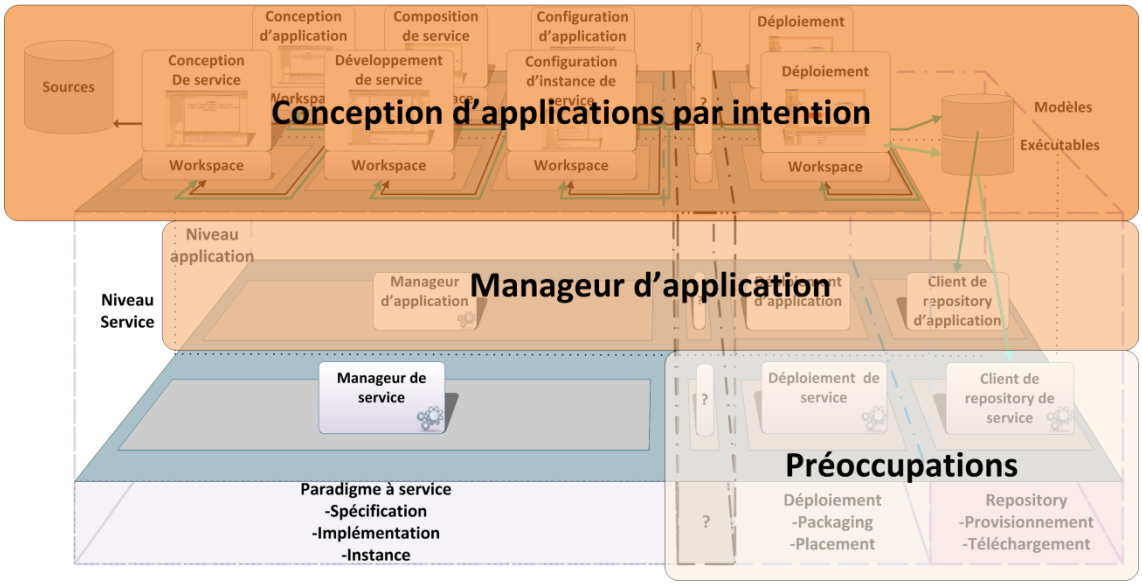


Figure 95 Architecture de l'environnement SAM

Au niveau de l'exécution, nous avons prototypé deux extensions : pour les unités de déploiement et pour les clients de dépôt, mais ils ne répondent que partiellement aux problèmes. De manière plus générale nous devrions aborder un ensemble de préoccupations plus large, allant de la sécurité aux aspects transactionnels, en passant par la prise en charge de la gestion mémoire et de calcul, pour déterminer jusqu'où notre mécanique **d'extension à d'autres préoccupations** peut s'étendre.

Comme nous l'avons vu précédemment, aucun des trois projets n'a abordé l'objectif fixé de l'approche globale qui est l'exécution et l'adaptation d'une application en fonction des évolutions du contexte d'exécution. Pour cela il nous faut un **manageur d'application** sensible à l'évolution de l'architecture en cours d'exécution, et qui adapte celle-ci en fonction d'un modèle d'application. Il faut par conséquent **un environnement de conception et de développement d'applications** pouvant être adapté à l'exécution.

Bibliographie

[AAAl02] Assaf Arkin, Sid Askary, Scott Fordin, Wolfgang Jekeli, Kohsuke Kawaguchi, David Orchard, Stefano Pogliani, Karsten Riemer, Susan Struble, Pal T. Nagy, Ivana Trickovic, and Sinisa Zimek. **Web Service Choreography Interface (WSCI)** 1.0. Technical report, 2002. http://www.w3.org/TR/wsci/.

[ACDAl03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, Sanjiva Weerawarana. **Business Process Execution Language (BPEL) for Web Services**, Version 1.1, may 2003, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf

[ACKM04] G. Alonso, F. Casati, H. Kuno, and H. Machiraju. **Web Services -Concepts, Architectures and Applications**. Springer Verlag, Berlin 2004.

[ADBAl99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. **Towards a better understanding of context and context-awareness**. In Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing, HUC '99, pages 304–307, Springer-Verlag. London, UK, 1999.

[AF01] L.F. Andrade et J.L. Fiadeiro. **Coordination technologies for web-services**. Dans Workshop on Object-Oriented Web Services (at OOPSLA), 2001.

[AFHAl99] Alonso, G.; Fiedler, U.; Hagen, C.; Lazcano, A.; Schuldt, H. & Weiler, N. **WISE: business to business e-commerce Research Issues on Data Engineering: Information Technology for Virtual Enterprises**, 1999. RIDE-VE '99. Proceedings., Ninth International Workshop on, 1999, 132-139

[Ars04] A. Arsanjani**. Service-oriented Modeling and Architecture: How to Identify, Specify, and Realize Services for Your SOA**. 2004, http://www.ibm.com/developerworks/webservices/library/ws-soa-design1/

[Ars08] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley. **Soma: a method for developing service-oriented solutions**. IBM Syst. J., 47(3):377–396, 2008.

[AtWo92] D. Perry and A.L. Wolf, **Foundations for the study of software architecture**, SIGSOFT Software Engineering Notes, vol. 17, no. 4, 1992, pp. 40-52.

[Ba00] F. Bachman et Al. Volume ii : **Technical concepts of component-based software engineering**. Technical report, Rapport technique ESC-TR-2000-007, Software Engineering Institute, May 2000.

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. **Models@run.time**. Computer, 42:22–27, 2009.

[BBHAl05] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Noel de Palma, Vivien Quema and Jean-Bernard Stefani. **Architecture-Based Autonomous Repair Management: Application to J2EE Clusters.** Autonomic Computing, International Conference on, IEEE Computer Society, 2005, 0, 369-370

[BCAl01] Blair, G.; Coulson, G.; Andersen, A.; Blair, L.; Clarke, M.; Costa, F.; Duran-Limon, H.; Fitzpatrick, T.; Johnston, L.; Moreira, R. & others. T**he design and implementation of Open ORB 2.** IEEE Distributed Systems Online. 2001 vol. 2, no. 6, pp 1-40

[BCS02] Bruneton, E., Coupaye, T. & Stefani, J.. **Recursive and dynamic software composition with sharing**. 2002

[BCS04] Bruneton, E., Coupaye, T. et Stefani, J.B., **The Fractal Composition Framework** Version 2.0-3, Object Web Consortium, July 2004.

[BDHT06] Sara Bouchenak, Noel De Palma, Daniel Hagimont and Christophe Taton. **Autonomic Management of Clustered Applications**. Cluster Computing, IEEE International Conference on, IEEE Computer Society, 2006, 0, 1-11

[BDO05] A. Barros, M. Dumas, and P. Oaks, **A critical overview of the web services choreography description language (ws-cdl)**. BPTrends, March 2005.

[Bez05] Jean Bézivin. **On the unification power of models.** Software and Systems Modeling, 4(2):171–188, May 2005

[BG01] Jean Bézivin and Olivier Gerbé. **Towards a precise definition of the omg/mda framework**. In ASE '01: Proceedings of the 16th IEEE international conference on Automated software engineering, page 273, Washington, DC, USA, 2001. IEEE Computer Society.

[BJV04] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. **On the need for megamodels**. In Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004.

[BLAl00] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro, **Service-based software : the future for flexible software**, apsec (2000), 214.

[Bonjour-S] Apple. **Bonjour Protocol Specifications**.
http://developer.apple.com/networking/bonjour/specs.html

[Bou08] Johann Bourcier. **Auto-Home : une plate-forme pour la gestion autonomique d'applications pervasives**. PhD thesis, Université Joseph Fourier, Grenoble, November 2008.

[Box97] Don Box. **Essential COM**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[BSSG08] André Bottaro, Eric Simon, Stéphane Seyvoz, and Anne Gérodolle. **Dynamic web services on a home service platform**. In AINA '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications, pages 378–385, Washington, DC, USA, March 2008. IEEE Computer Society.

[CADSE] Equipe Adèle. **CADSE** site. http://cadse.imag.fr/

[CCMS06] CORBA **Component Model Specification version 4.0**, OMG,
http://www.omg.org/spec/CCM/4.0/, 2006

[CeRi04] Humberto Cervantes and Richard Hall. A**utonomous adaptation to dynamic availability using a service-oriented component model**. In ICSE, pages 614–623. IEEE Computer Society, May 2004.

[CerT04] Humberto Cervantes. **Vers un modèle à composants orienté services pour supporter la disponibilité dynamique**. PhD thesis, Université Joseph Fourier, Grenoble, March 2004.

[CFAl98] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner, André van der Hoek, Alexander L. Wolf, Andre Van Der, Er L. Wolf, and Er L. Wolf. **A characterization framework for software deployment technologies**. Technical report, 1998.

[CGFP09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano. **Autonomic computing through reuse of variability models at runtime: The case of smart homes**. Computer, 42:37–43, 2009

[Cha04] Chappell, D. **Enterprise Service Bus**. O'Reilly Media, Inc., ISBN:0596006756 2004

[CheT08] Shang-Wen Cheng. **Rainbow: cost-effective software architecture-based self-adaptation**. Thesis of Carnegie Mellon University, 2008

[CHGAl04] Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl and Peter Steenkiste. **Rainbow: Architecture-based self-adaptation with reusable infrastructure**. IEEE Computer, 2004, 37, 46-54

[CNW01] Francisco Curbera, William A. Nagy, and Sanjiva Weerawarana. **Web services: Why and how**. In OOPSLA 2001 Workshop on Object-Oriented Web Services. ACM, August 2001

[Corb08] Object Management Group (OMG™). **CORBA 3.1**, 2008. http://www.omg.org/spec/CORBA/3.1/.

[DCD06] Mikaël Desertot, Humberto Cervantes, and Didier Donsez. **Frogi: Fractal components deployment over osgi**. In Software Composition, pages 275–290. Springer, March 2006.

[Dear07] Alan Dearle. **Software Deployment, Past, Present and Future** Future of Software Engineering, 2007. FOSE '07, Future of Software Engineering, 2007. FOSE '07, 2007, 269-284

[DES02] Frédéric Duclos, Jacky Estublier and Rémy Sanlaville. **Une Machine à Objets Extensibles pour la Séparation des Préoccupations**. In Proceeding Journées Systèmes à Composants Adaptables et extensibles , Grenoble, France, 2002-10-01.

[DieT10] Idrissa Dieng. SELECTA - **Une approche de construction d'applications par composition de services**. PhD thesis, Université de Grenoble, May 2010.

[DPWS-S] Shannon Chan, Dan Conti, Chris Kaler, Thomas Kuehnel, Alain Regnier, Bryan Roe, Dale Sather , Jeffrey Schlimmer, Hitoshi Sekine, Jorgen Thelin, Doug Walter, Jack Weast, Dave Withehead, Don Wright and Yevgniy Yarmosh. **Devices Profile for Web Services**. Microsoft Corporation, February 2006. http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf

[EDL10] Jacky Estublier, Idrissa Dieng, and Thomas Lévêque. **Software Product Line Evolution: The Selecta System**. In PLEASE ’10: Proceedings of the 2010 ICSE Workshop on Product Line Approaches in Software Engineering, pages 32–39, New York, NY, USA, May 2010. ACM

[EDSG09] Jacky Estublier, Idrissa Dieng, Eric Simon, and German Vega. **Flexible composites and automatic component selection for service-based applications**. In Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Milan, Italy, May 2009. INSTICC Press.

[EDSM10] Jacky Estublier, Idrissa Dieng, Eric Simon, and Diana Moreno-Garcia. **Opportunistic computing experience with the SAM platform**. In Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS '10, pages 1–7, New York, NY, USA, 2010. ACM

[EHL07] C. Escoffier, R. S. Hall, and P. Lalanda. **iPOJO: an extensible service-oriented component framework**. In Proc. IEEE International Conference on Services Computing SCC 2007, pages 474–481, 9–13 July 2007

[EJB] Oracle (Sun Microsystems). **Enterprise JavaBeans Technology**.
http://www.oracle.com/technetwork/java/index-jsp-140203.html

[EJB3.0] Oracle (Sun Microsystems). **Enterprise Java Beans version 3.0**
http://www.oracle.com/technetwork/java/docs-135218.html

[ELV09] Jacky Estublier, Thomas Leveque, and German Vega. **Evolution Control in MDE Projects: Controlling Model and Code Co-evolution**. In Farhad Arbab and Marjan Sirjani, editors, Fundamentals of Software Engineering, volume 5961 of Lecture Notes in Computer Science, pages 431–438, Berlin, Heidelberg, 2009. Springer Berlin / Heidelberg.

[End04] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Poel Krogdahl, Min Luo, and Tony Newling. **Patterns: Service-Oriented Architecture and Web Services**. IBM Redbooks, July 2004 http ://www.redbooks.ibm.com/redbooks/pdfs/sg246303.pdf, 2004.

[EscT08] Clément Escoffier. iPOJO : **Un modèle à composant à service flexible pour les systèmes dynamiques**. PhD thesis, Université Joseph Fourier, Grenoble, December 2008.

[EVLL08] Jacky Estublier, German Vega, Thomas Leveque and Philippe Lalanda. **Domain specific engineering environments**. In Asia-Pacific Software Engineering Conference, Dec. 2008. APSEC, IEEE, 2008, 553-560

[Fav04] Jean-Marie Favre. **Towards a Basic Theory to Model Model Driven Engineering**. 3rd Workshop in Software Engineering. 2004.

[FEB06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. **L'Ingénierie Dirigée par les Modèles : au-delà du MDA**. Informatique et Systèmes d'Information. Hermes Science, lavoisier edition, Février 2006.

[FHLAl09] Nicolas Ferry, Vincent Hourdin, Stephane Lavirotte, Gaetan Rey, Jean-Yves Tigli and Michel Riveill. **Models at Runtime: Service for Device Composition and Adaptation**. 4th International Workshop Models@run.time at Models 2009(MRT'09), 2009, 51-60

[FHSAl06] Jacqueline Floch, Svein Hallsteinsen, Erlen Stav, Frank Eliassen, Ketil Lund, and Eli Gjorven. **Using architecture models for runtime adaptability**. Software, IEEE, 23(2):62–70, 2006.

[FR07] Robert France and Bernhard Rumpe. **Model-driven development of complex software: A research roadmap**. In FOSE '07: 2007 Future of Software Engineering, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society

[Fra04] ObjectWeb. **The Fractal Project**, 2004. http://fractal.objectweb.org/.

[FugA00] A. Fuggetta **Software process: a roadmap**. ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pages 25-34 New York, NY, USA, 2000. ACM,

[Gane03] A. G. Ganek and T. A. Corbi. **The dawning of the autonomic computing era**. IBM Syst. J., 42(1):5–18, 2003.

[Hall99] Richard Scott Hall. **Agent-based Software Configuration and Deployment**. PhD thesis, 1999

[HC01] George T. Heineman and William T. Councill. **Component-Based Software Engineering**: Putting the Pieces Together (ACM Press). Addison-Wesley Professional, June 2001.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. **Separation of concerns**. Technical report, 1995. http://ftp.ccs.neu.edu/pub/people/crista/publications/techrep95/separation.pdf

[Horn01] Paul Horn, **Autonomic Computing: IBM's Perspective on the State of Information Technology**. IBM, 2001.

[HTLRR08] Vincent Hourdin, Jean-Yves Tigli, Stephane Lavirotte, Gaetan Rey and Michel Riveill. **SLCA, composite services for ubiquitous computing**. Proceedings of the 5th International Conference on Mobile Technology, Applications and Systems(Mobility) , long paper, 2008, 8

[iPOJO-s]. **iPOJO site**. http://felix.apache.org/site/apache-felix-ipojo.html

[IRB94] Jadwiga Indulska, Kerry Raymond, and Mirion Bearman. **A type management system for an ODP trader**. In Proceedings of the IFIP TC6/WG6.1 International Conference on Open Distributed Processing II, pages 169–180, Amsterdam, The Netherlands, 1994. North-Holland Publishing Co.

[Jack02] Michael Jackson. **Some basic tenets of description**. Software and System Modeling, 1(1):5–9, 2002.

[JBS97] Sun Microsystems. **Java beans specification.** 1997
http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp

[Jini] Jini.org. **Jini** . http://www.jini.org/

[JMS99] Hapner, M.; Burridge, R. & Sharma, R. **Java™Message Service specification -version 1.0.2**. SUN. November 1999 http://dlc.sun.com/pdf/816-5904-10/816-5904-10.pdf

[Keph03] J. O. Kephart and D. M. Chess. **The vision of autonomic computing.** Computer, 36(1):41–50, Jan. 2003.

[Keph05] J. O. Kephart. Research challenges of autonomic computing. In Proc. 27th International Conference on Software Engineering ICSE 2005, pages 15–22, 15–21 May 2005.

[KHHAl01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353, London, UK, 2001. Springer-Verlag.

[KrMa07] Kramer, J. & Magee, J. S**elf-Managed Systems: an Architectural Challenge** Future of Software Engineering, 2007. FOSE '07, Future of Software Engineering, 2007. FOSE '07, IEEE Computer Society, 2007, 259-268

[KSKC04] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. **Composing adaptive software.** Computer, 37:56–64, 2004. IEEE Computer Society,

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. **MDA Explained: The Model Driven Architecture: Practice and Promise.** Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[LBTA10] Grzegorz Lehmann, Marco Blumendorf, Frank Trollman and Sahin Albayrak. **Meta-Modeling Runtime Models** Proceedings of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway , CEUR-WS.org , 2010 , 641, 1-12

[Lee00] M. H. Lee. **Model-based reasoning: a principled approach for software engineering**. Software - Concepts & Tools, 19(4):179– 189, 2000

[LevT10] Thomas Lévêque. **Définition et contrôle des politiques d'évolution dans les projets logiciels**. PhD thesis, Université de Grenoble, June 2010.

[Lin00] David S. Linthicum. E**nterprise application integration**. Addison-Wesley Longman Ltd., Essex, UK, UK, 2000

[Lud03] Jochen Ludewig. **Models in software engineering an introduction**. Software and Systems Modeling, 2:5–14, 2003. 10.1007/s10270-003-0020-3.

[MBJFS09] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Frank Fleurey and Arnor Solberg. **Models@Run.time to Support Dynamic Adaptation Computer**, IEEE Computer Society, 2009, 42, 44-51

[MDA-O] OMG. **Model Driven Architecture**. http://www.omg.org/mda/

[MFBC10] Pierre-Alain Muller, Fédéric Fondement, Benoit Baudry, and Benoit Combemale. **Modeling modeling modeling**. SOSYM, 2010.

[MIND09] MINDC User Guide version 0.2-alpha-2 (Draft). ObjectWeb 2, 2009

[MN97] Theo Dirk Meijler and Oscar Nierstrasz. **Beyond Objects: Components, cooperative information systems**: trends and directions. edition, 1997.

[MOF09] OMG. **MOF : MetaObject Facility**. 2009, http://www.omg.org/mof/

[Myer00] Brad A. Myers. **Using multiple devices simultaneously for display and control**. Special issue of IEEE Personal Communications, Smart Spaces and Environments, 7:62–65, 2000

[nato68] NATO **Software Engineering Conference**, 1968

[OASI06] OASIS. **Reference Model for Service Oriented Architecture**. October 2006. http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf.

[OMGa06] OMG. **Deployment and Configuration of Component-based Distributed Applications Specification** Version 4.0. April 2006. http://www.omg.org/docs/formal/06-04-02.pdf

[Ore99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. **An architecture-based approach to self-adaptive software**. IEEE Intelligent Systems, 14:54–62, 1999

[OSGi-S] OSGi™ Alliance. **OSGi site**. http://www.osgi.org/

[OSGia] OSGi Alliance. **OSGi Service Platform Core Specification release 4.2**. June 2009. http://www.osgi.org/download/r4v42/r4.core.pdf

[OSGib] OSGi Alliance. **OSGi Service Platform Service Compendium release 4.2**. August 2009. http://www.osgi.org/download/r4v42/r4.cmpn.pdf

[OSGic] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. **An architecture-based approach to self-adaptive software**. IEEE Intelligent Systems, 14:54–62, 1999.

[PaGe03] M. P. Papazoglou and D. Georgakopoulos. **Service oriented computing**. Communications of the ACM, 46:24–28, October 2003.

[PaHe06] Michael P. Papazoglou and Willem-Jan V. Heuvel. **Service-oriented design and development methodology**. International Journal of Web Engineering and Technology, 2(4):412+, 2006

[Pap03] M. P. Papazoglou. **Service-oriented computing: concepts, characteristics and directions**. Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on, pages 3–12, 2003

[Par72] D.L. Parnas. **On the criteria to be used in decomposing systems into modules**. Communications of the ACM, 15(12), Decembre 1972, 1053 -1058.

[PDE09] Gabriel Pedraza, Idrissa Dieng and Jacky Estublier. **FOCAS: An Enginering Environment for Service-Based Applications**. In Proceeding Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE), Milan, Italy, 2009-05-06.

[PE08] Gabriel Pedraza and Jacky Estublier. **An Extensible Services Orchestration Framework through Concern Composition.** In Proceeding International Workshop on Non-functional System Properties in Domain Specific Modeling Languages (NFPDSML 2008), Toulouse, France, 2008-09-28.

[PE09] Gabriel Pedraza and Jacky Estublier. **Distributed orchestration versus choreography: The FOCAS approach**. In Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes, ICSP '09, pages 75–86, Berlin, Heidelberg, 2009. Springer-Verlag

[PedT09] Gabriel Pedraza. **FOCAS : un canevas extensible pour la construction d'applications orientées procédé**. PhD thesis, Université Joseph Fourier, Grenoble, November 2009

[Pel03] Chris Peltz. **Web services orchestration and choreography**.Computer, 36:46–52, 2003

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. **Service oriented architectures: approaches, technologies and research issues**. The VLDB Journal, 16(3):389–415, 2007

[PTDL07] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar and Frank Leymann **Service-oriented computing: State of the art and research challenges.** Computer, 40(11):38–45, Nov. 2007.

[PTDLK06] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, Frank Leymann, and Bernd J. Krämer. S**ervice-oriented computing research roadmap**. In Dagstuhl Seminar Proceedings 05462, pages 1–29, April 2006

[RAJM01] Tyler Jewell Ed Roman, Scott W. Ambler and Floyd Marinescu. **Mastering Enterprise JavaBeans**, 2nd Edition. John Wiley & Sons, 2001.

[Seid03] Ed Seidewitz. **What Models Mean**. IEEE Software, 20(5):26–32, 2003

[SSDP99] Paul Leach Ye Gu Yaron Y. Goland, Ting Cai and Shivaun Albright, **Simple Service Discovery Protocol**, IETF Draft draft-cai-ssdp-v1-03.txt, October 28, 1999. http://www.ietf.org/internetdrafts/draft-cai-ssdp-v1-03.txt

[Szy02] Clemens Szyperski. **Component Software: Beyond Object- Oriented Programming (2nd Edition).** Addison-Wesley Professional, 2 edition, November 2002

[Tay98] David A. Taylor. **Object technology (2nd ed.): a manager's guide**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998

[TBB03] M. Turner, D. Budgen, and P. Brereton. **Turning software into a service**. Computer, 36(10):38–44, October 2003.

[TIB05] Chouki Tibermacine. **Un méta-modèle pour la description de contraintes architecturales sur l'évolution des composants**. In Proceedings of the workshop on software evolution (SE'05), held in conjunction with LMO'05, Bern, Switzerland, March 2005

[TLRHR09] Jean-Yves Tigli, Stephane Lavirotte, Gaetan Rey, Vincent Hourdin and Michel Riveill. **Lightweight Service Oriented Architecture for Pervasive Computing**. International Journal of Computer Science Issues (IJCSI), 2009, 4, 1-9

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Jr. N **Degrees of separation: multi-dimensional separation of concerns**. In ICSE '99: Proceedings of the 21st international conference on Software engineering, pages 107–119, New York, NY, USA, 1999. ACM.

[UPnP] UPnP forum. **Universal Plug'n Play** site. http://upnp.org/

[UPnP-S] UPnP forum. **UPnP Specifications**. http://upnp.org/sdcps-and-certification/standards/

[VSG10] Thomas Voge, Andreas Seibel Holger Giese. **Toward Megamodels at Runtime.** Proceedings of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway , CEUR-WS.org , 2010 , 641 , 13-24

[War79] Wartofsky, M. W., Cohen, R. S. (ed.) **Models: Representation and the Scientific Understanding** Springer, 1979

[WD01] Roel Wuyts and Stéphane Ducasse. **Composition languages for black-box components**. In Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components, pages 33–36, October 2001.

[Weis91] Mark Weiser. **The computer for the 21st century**. Scientific American, 265(3):66–75, January 1991

[WS-S] W3C. **Web of Services specifications**. http://www.w3.org/standards/webofservices/

[WSD09] OASIS. **Web Services Dynamic Discovery (WS-Discovery) Version 1.1**. OASIS Standard, 1 July 2009, http://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.pdf

[Yan03] Jian Yang. **Web service componentization.** Communication. ACM, 46(10):35–40, 2003

[ZhZh09] Liang-Jie Zhang and Jia Zhang. **Design of service component layer in soa reference architecture**. Computer Software and Applications Conference, Annual International, 1:474–479, 2009

Annexe A - Espace, Environnement ou Contexte

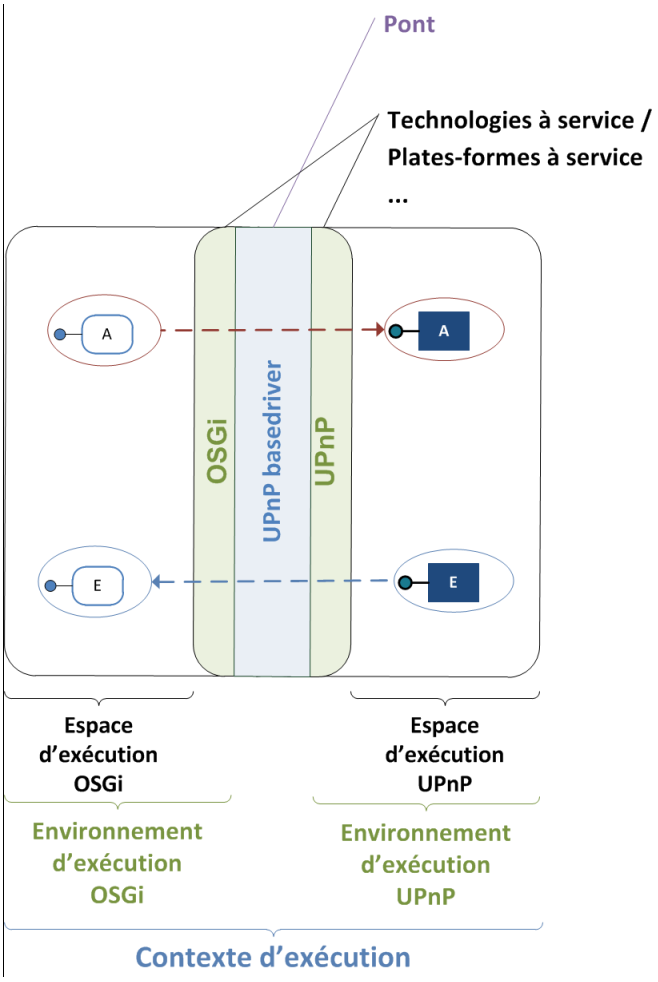


Figure 96 Espace, Environnement ou Contexte

- Espace d'exécution : elle correspond à l'ensemble des éléments métiers d'une technologie donnée à l'exécution ; par exemple l'ensemble des services métiers en interaction dans OSGi. L'espace d'exécution est lié une technologie à l'exécution, de même qu'un espace de travail (*workspace*) est lié à un IDE.
- Environnement d'exécution : il correspond à l'ensemble des éléments métiers et des éléments de l'infrastructure ; c'est-à-dire le SOA et les services.
- Contexte d'exécution : est l'ensemble de tous les aspects de l'exécution.

Annexe B - Lexique et définitions

AM : signifie Abstract Machine. Il désigne un environnement.

AM* : désigne un ensemble d'AM interconnectés, soit un environnement réparti.

SAM : signifie *Service Abstract Machine*. Il désigne l'approche globale.

SAM-CORE : désigne le méta-modèle des composants orientés service utilisé dans l'approche SAM. Notons cependant que SAM-CORE peut désigner le méta-modèle de l'approche globale, comme celui utilisé à l'exécution. Les sections 1 et 2 du Chapitre 3 - SAM CORE désigne méta-modèle de l'approche globale ; puis dans la suite le méta-modèle à l'exécution.

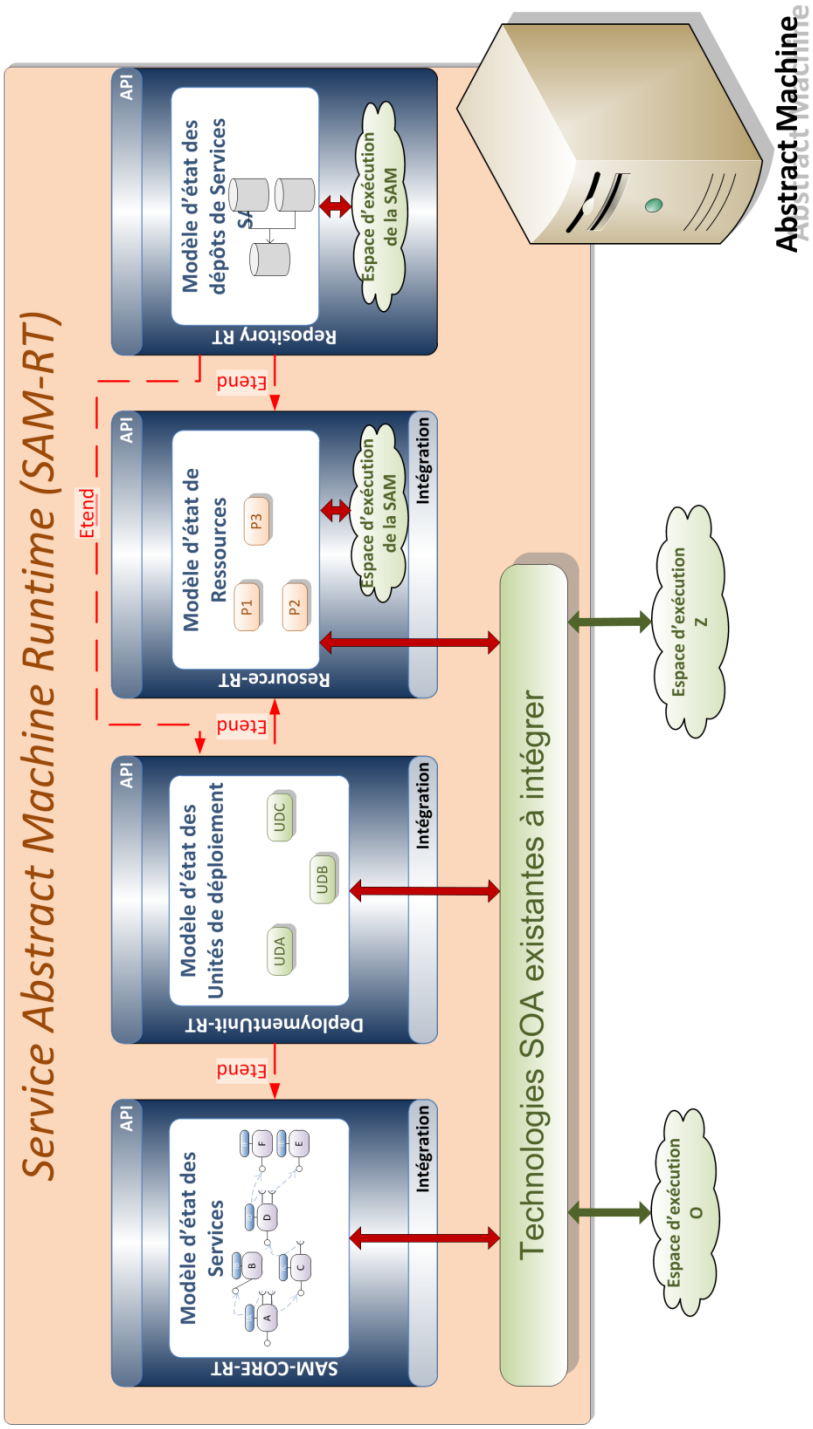
SAM-CORE-RT : signifie *SAM CORE runtime*. Il désigne l'AM spécialisé pour être un environnement d'exécution d'architectures orientées service. L'architecture est définie par le méta-modèle *SAM CORE*.

SAM-CORE-RT* : désigne le sous-ensemble d'AM* où chaque AM est spécialisé en SAM-CORE-RT.

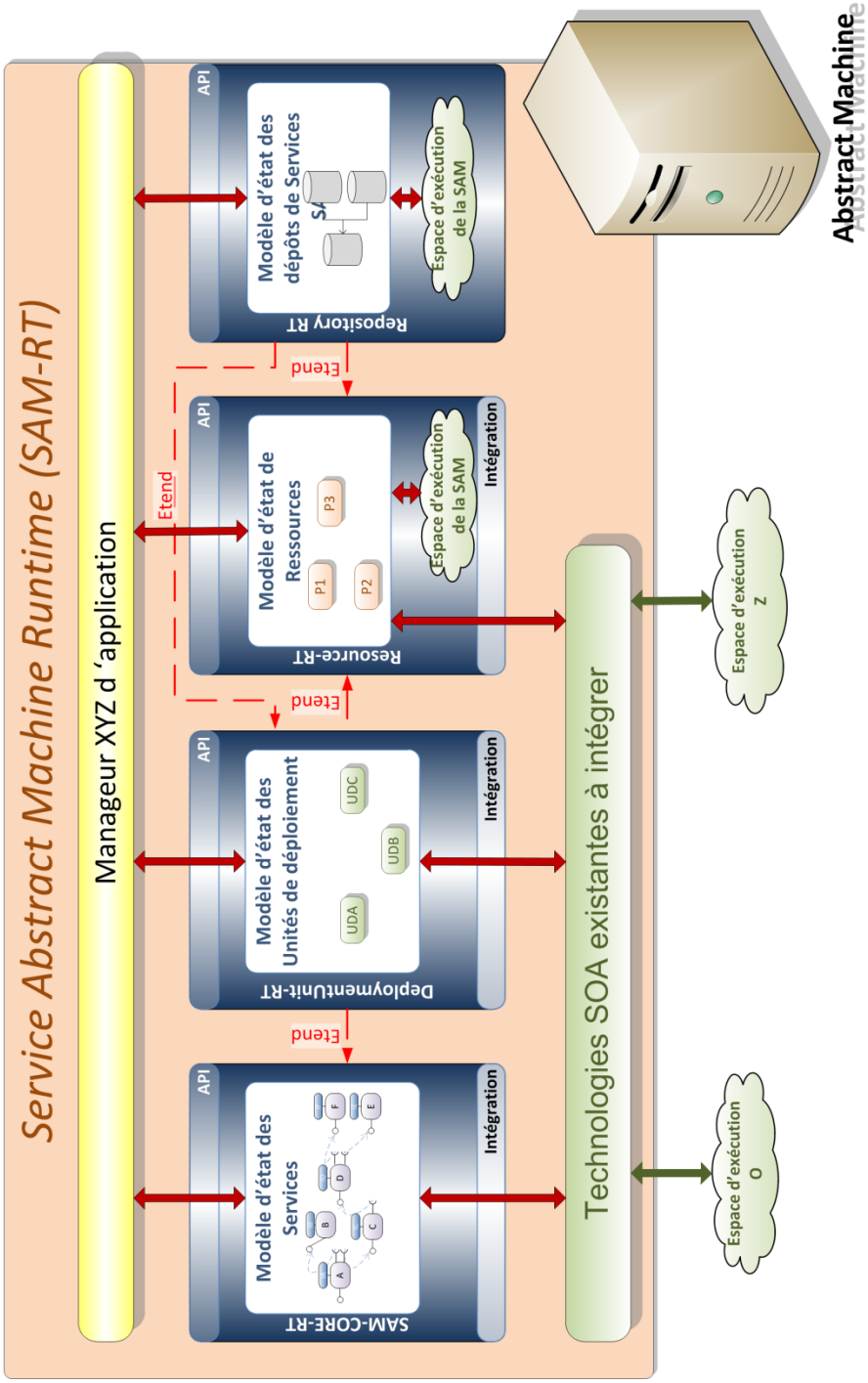
SAM-RT : signifie *SAM Runtime*. Il désigne de manière générale l'environnement une AM spécialisé par SAM-CORE-RT et par zéro ou plusieurs autres extensions, comme par exemple *DeploymenUnit Runtime*.

SAM-RT* : désigne le sous-ensemble d'AM* où chaque AM est spécialisé en SAM-RT

Annexe C - Extensibilité



Annexe D - Interaction *Manager/Runtime*



Annexe E - Règle de critère qualité (SONAR)

Règle	Type	Niveau	AM	SAM	SCM
Anon Inner Length	Maintainability	MAJOR			
Avoid Array Loops	Efficiency	MAJOR			
Avoid Assert As Identifier	Portability	MAJOR			
Avoid Calling Finalize	Usability	MAJOR			
Avoid Catching NullPointerException	Reliability	MAJOR	5	6	2
Avoid Catching Throwable	Reliability	CRITICAL			
Avoid Decimal Literals In Big Decimal Constructor	Reliability	MAJOR			
Avoid Duplicate Literals	Maintainability	MAJOR	1	6	2
Avoid Enum As Identifier	Portability	MAJOR			
Avoid Instanceof Checks In Catch Clause	Maintainability	MINOR	11		
Avoid Print Stack Trace	Usability	MAJOR	23	26	9
Avoid Rethrowing Exception	Maintainability	MAJOR	2		
Avoid Throwing Null Pointer Exception	Usability	MAJOR	12	3	
Avoid Throwing Raw Exception Types	Maintainability	MAJOR			
Big Integer Instantiation Efficiency	Efficiency	MAJOR			
Boolean Expression Complexity	Maintainability	MAJOR	2	1	
Boolean Instantiation	Efficiency	MAJOR			
Broken Null Check	Reliability	CRITICAL			
Class Cast Exception With To Array	Reliability	MAJOR			
Clone Throws Clone Not Supported Exception	Reliability	MAJOR			
Clone method must implement Cloneable	Reliability	MAJOR			
Close Resource	Reliability	MAJOR			
Collapsible If Statements	Maintainability	MINOR	51	1	
Compare Objects With Equals	Reliability	MAJOR			
Constant Name	Usability		1	1	
Constructor Calls Overridable Method	Reliability	MAJOR			
Cyclomatic Complexity	Maintainability	MAJOR	17	5	10
Default Comes Last	Usability	MAJOR			
Design For Extension	Reliability	MINOR	201	224	174
Dont Import Java Lang	Maintainability	MINOR			
Dont Import Sun	Portability	MINOR			
Double Checked Locking	Reliability	MAJOR			
Empty Finalizer	Maintainability	MAJOR			
Empty Finally Block	Maintainability	CRITICAL			
Empty If Stmt	Maintainability	CRITICAL	3	1	8

Empty Statement	Maintainability	MINOR		1	
Empty Static Initializer	Maintainability	MAJOR			
Empty Switch Statements	Maintainability	MAJOR			
Empty Synchronized Block	Maintainability	CRITICAL			
Empty Try Block	Maintainability	MAJOR			
Empty While Stmt	Maintainability	CRITICAL			
Equals Hash Code	Reliability	CRITICAL			
Equals Null	Reliability	CRITICAL			
Exception As Flow Control	Usability	MAJOR			
Final Class	Usability	MAJOR	2	1	
Final Field Could Be Static	Efficiency	MINOR			
Finalize Does Not Call Super Finalize	Reliability	MAJOR			
Finalize Overloaded	Reliability	MAJOR			
For Loops Must Use Braces	Usability	MAJOR			
Hidden Field	Usability	MAJOR			
Hide Utility Class Constructor	Efficiency	MAJOR		1	3
Idempotent Operations	Efficiency	MAJOR			
If Else Stmts Must Use Braces	Usability	MAJOR	4		
If Stmts Must Use Braces	Usability	MAJOR	1		2
Illegal Throws	Maintainability	MAJOR			
Inefficient String Buffering	Efficiency	MAJOR			
Inner Assignment	Usability	MAJOR			
Instantiation To Get Class	Usability	MAJOR			
Integer Instantiation	Portability	MAJOR	2		
Local Final Variable	Usability	MAJOR			
Local Variable Name	Usability	MAJOR	4		14
Loose coupling	Maintainability	MAJOR			
Magic Number	Reliability	MINOR	72	1	
Member Name	Usability	MAJOR	2		2
Method Name	Usability	MAJOR			5
Missing Static Method In Non Instantiatable Class	Maintainability	MAJOR			
Modifier Order	Usability	MINOR	28	6	11
Naming - Avoid dollar signs	Usability	MINOR			
Naming - Class naming conventions	Usability	MAJOR			
Naming - Method with same name as enclosing class	Usability	MAJOR			
Naming - Suspicious Hashcode method name	Usability	MAJOR			
Naming - Suspicious constant field name	Usability	MAJOR	3		1
Naming - Suspicious equals method name	Usability	CRITICAL			
Ncss Method Count	Maintainability	MAJOR	8	1	1
Ncss Type Count	Maintainability	MAJOR			
Package Name	Usability	MAJOR			
Parameter Assignment	Usability	MAJOR	4		2

Parameter Name	Usability	MAJOR			
Preserve Stack Trace	Maintainability	MAJOR	4		1
Redundant Modifier	Maintainability	MINOR	104	183	53
Redundant Throws	Maintainability	MINOR	20	26	5
Replace Enumeration With Iterator	Portability	MAJOR			
Replace Hashtable With Map	Portability	MAJOR			
Replace Vector With List	Portability	MAJOR			
Security - Array is stored directly	Reliability	CRITICAL	4		
Signature Declare Throws Exception	Maintainability	MAJOR	21	2	
Simplify Boolean Expression	Maintainability	MAJOR			
Simplify Boolean Return	Maintainability	MAJOR	1		1
Simplify Conditional	Maintainability	MAJOR			
Singular Field	Maintainability	MINOR	5	1	7
Static Variable Name	Usability	MAJOR	1		1
String Buffer Instantiation With Char	Reliability	MAJOR			
String Instantiation	Efficiency	MAJOR			
String Literal Equality	Reliability	MAJOR	2		
String To String	Maintainability	MAJOR			
System Println	Usability	MAJOR		1	5
Unconditional If Statement	Maintainability	CRITICAL			
Unnecessary Case Change	Efficiency	MINOR			
Unnecessary Local Before Return	Efficiency	MAJOR			1
Unused Imports	Maintainability	INFO	1	12	4
Unused Modifier	Maintainability	INFO	102	183	5
Unused Null Check In Equals	Maintainability	MAJOR			
Unused Private Field	Maintainability	MAJOR	9	6	8
Unused formal parameter	Maintainability	MAJOR	5	1	4
Unused local variable	Maintainability	MAJOR	4		3
Unused private method	Maintainability	MAJOR			
Use Array List Instead Of Vector	Efficiency	MAJOR			
Use Arrays As List	Efficiency	MAJOR			
Use Correct Exception Logging	Maintainability	MAJOR			
Use Index Of Char	Efficiency	MAJOR	1		
Use String Buffer Length	Efficiency	MINOR			
Useless Operation On Immutable	Reliability	CRITICAL	1		
Useless Overriding Method	Maintainability	MAJOR			
Useless String Value Of	Efficiency	MINOR			
Visibility Modifier	Maintainability	MAJOR	7		10
While Loops Must Use Braces	Usability	MAJOR			

Annexe F - Exemples de message du protocole de découverte

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Header>
    <am:hello xmlns:am="http://fr.imag.adele.am/1.2.0/message" at=
    "Mon Oct 19 11:10:09 CEST 2009" during="30000" version="1.2.0" >
      <am:from name="A" reference="http://14.24.32.124:8081" >
        <address value="http://14.24.32.124:8081" />
        <address value="http://10.0.2.4:8081" />
        <address value="http://192.168.0.2:8081" />
      </am:from>
    </am:hello>
  </env:header>
  <env:body>
    <properties>
      <property am.machine.name="A" />
      <property am.machine.address="http://14.24.32.124:8081" />
    </properties>
  </env:body>
</env:envelope>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Header>
    <am:isalive xmlns:am="http://fr.imag.adele.am/1.2.0/message" at="
    Mon Oct 19 11:10:09 CEST 2009" during="30000" version="1.2.0" >
      <am:from name="A" reference="http://14.24.32.124:8081" >
        <address value="http://14.24.32.124:8081" />
        <address value="http://10.0.2.4:8081" />
        <address value="http://192.168.0.2:8081" />
      </am:from>
    </am:isalive>
  </env:header>
  <env:body>
    <properties>
      <property am.machine.name="A" />
      <property am.machine.address="http://14.24.32.124:8081" />
    </properties>
  </env:body>
</env:envelope>
```

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope" >
  <env:Header>
    <am:bye xmlns:am="http://fr.imag.adele.am/1.2.0/message" at=
    "Mon Oct 19 11:10:09 CEST 2009"  version="1.2.0" >
      <am:from name="A" reference="http://14.24.32.124:8081" >
        <address value="http://14.24.32.124:8081" />
        <address value="http://10.0.2.4:8081" />
        <address value="http://192.168.0.2:8081" />
      </am:from>
    </am:bye>
  </env:header>
  <env:body/>
</env:envelope>
```

Annexe G - Liste des publications

- **Eric Simon**, Jacky Estublier and Diana Moreno. **Extensible and General Service-Oriented Platform: Experience with the Service Abstract Machine.** IEEE International Conference on Services Computing (SCC2010), IEEE Computer Society, 2010, 490-497 (Taux d'acceptation 18%)

- Jacky Estublier and **Eric Simon**. **Universal and Extensible Service-Oriented Platform Feasibility and experience: The Service Abstract Machine**. In *Proceedings Second IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications,* Seatle, USA, 2009-07-20.

- **[EDSG09]** Jacky Estublier, Idrissa Dieng, **Eric Simon**, and German Vega. **Flexible composites and automatic component selection for service-based applications**. In Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), Milan, Italy, May 2009. INSTICC Press. (Taux d'acceptation 30%)

- **[EDSM10]** Jacky Estublier, Idrissa Dieng, **Eric Simon**, and Diana Moreno-Garcia. **Opportunistic computing experience with the SAM platform**. In Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems, PESOS '10, pages 1–7, New York, NY, USA, 2010. ACM

- **[BSSG08]** André Bottaro, **Eric Simon**, Stéphane Seyvoz, and Anne Gérodolle. **Dynamic web services on a home service platform**. In AINA '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications, pages 378–385, Washington, DC, USA, March 2008. IEEE Computer Society. (Taux d'acceptation 30%)