VERS UNE INGENIERIE DE L’EVOLUTION LOGICIELLE : UNE APPROCHE ARCHITECTURALE. VERROUS, CONTRIBUTIONS ET PERSPECTIVES.

Dalila Tamzalit

▶ To cite this version:

Dalila Tamzalit. VERS UNE INGENIERIE DE L’EVOLUTION LOGICIELLE : UNE APPROCHE ARCHITECTURALE. VERROUS, CONTRIBUTIONS ET PERSPECTIVES.. Génie logiciel [cs.SE]. Université de Nantes, 2020. tel-02967449

HAL Id: tel-02967449
https://hal.science/tel-02967449v1

Submitted on 22 Oct 2020

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.

HABILITATION A DIRIGER DES RECHERCHES

Dalila TAMZALIT

Dalila-Tamzalit.com

VERS UNE INGENIERIE DE L'EVOLUTION LOGICIELLE : UNE APPROCHE ARCHITECTURALE

VERROUS, CONTRIBUTIONS ET PERSPECTIVES

– Soutenance : 26 juin 2020 –

Composition du jury

Président du jury

Claude Jard (Professeur, LS2N / Université de Nantes)

Rapporteurs

Yamine Aït-Ameur (Professeur, IRIT / INPT-ENSEEIHT Toulouse)

Marianne Huchard (Professeure, LIRMM / Université Montpellier)

Houari Sahraoui (Professeur, DIRO/ Université de Montréal)

Examinateurs

Mireille Blay-Fornarino (Professeure, I3S / Université de Nice)

Xavier Blanc (Professeur, LaBRI / Université de Bordeaux)

Mourad Oussalah (Professeur, LS2N / Université de Nantes)

Yves Caseau (Directeur des Systèmes d'Information/ Michelin Clermont-Ferrand)

LS2N UMR 6004 **IUT de Nantes**

« La connaissance n'est que de l'eau stagnant au fond d'un vieux vase si on ne la laisse pas couler… »

Elif Shafak

Résumé

L'intitulé de mon document « *vers une ingénierie de l'évolution logicielle : une approche architecturale* » met en évidence le contexte des travaux présentés : la problématique générale de l'évolution des architectures logicielles et leurs descriptions. Le fil conducteur de mes travaux de recherche est de prôner une Ingénierie de l'évolution des architectures logicielles, en termes de fondements, modèles et méthodologies. Le terme *évolution* s'entend au sens large : migration, adaptation, refactoring, rétro-conception, restructuration, maintenance... Mes travaux de recherche s'articulent autour de la spécification et la formalisation : (i) des architectures logicielles évolutives et (ii) des architectures de l'évolution logicielle. La principale ligne directrice est d'identifier et de cerner les caractéristiques intrinsèques de systèmes afin de spécifier et gérer leur évolution indépendamment de leur fonctionnement. Le principe adopté est un principe de séparation stricte des préoccupations de l'évolution des autres préoccupations des architectures logicielles. L'approche adoptée est une approche par abstraction (métamodèles, architectures logicielles). En couplant ces deux caractéristiques, il a été possible de réifier différentes préoccupations liées à l'évolution sous forme de concepts statiques (modèles) et dynamiques (processus) à un haut niveau d'abstraction, permettant une représentativité aux niveaux stratégiques de décision, tout en offrant une assistance aux architectes. Ces travaux ont concerné différents paradigmes d'architectures logicielles : depuis l'objet durant ma thèse, aux composants en passant par la SOA et en considérant les styles architecturaux SaaS mutualisé et micro-services. Ces travaux ont été menés au travers d'encadrement et de co-encadrement doctoral de 9 thèses et 9 Master2, ainsi que des collaborations avec des confrères français, européens, internationaux et avec des entreprises.

Ce document présente une partie de mes travaux de recherche, ceux portant sur et pour les problématiques d'évolution des architectures logicielles à base de composants et autour de la notion de service[1], notamment les styles architecturaux SaaS mutualisé et micro-service. La raison est double : ces travaux constituent l'essentiel de mes activités de recherche et mes projets futurs s'inscrivent dans leur continuité.

L'idée essentielle de ces travaux est d'architecturer l'évolution logicielle par abstraction et modélisation. De 2002 à 2010, ils ont porté sur les architectures logicielles à base de composants au travers de leurs langages de description, les ADLs (*Architecture Description Languages*). Les principales contributions ont été de proposer un modèle générique pour l'évolution structurelle d'architectures logicielles, un modèle de capitalisation de savoir-faire d'évolutions architecturales, une approche d'extraction d'architectures logicielles à partir de systèmes objets patrimoniaux ainsi que des patrons d'évolution architecturale. De 2010 à aujourd'hui, mes travaux ont porté sur les architectures logicielles autour de la notion de service (SOA, SaaS, micro-services). Les principales contributions ont été de proposer un modèle d'externalisation de la gestion de la variabilité d'applications SaaS mutualisées, une approche SOA de restructuration et d'adaptation de workflows d'entreprise ainsi que des retours d'expérience de migration de systèmes monolithes vers des architectures micro-services.

Chacun de ces travaux est présenté selon son contexte, les verrous scientifiques qu'il traite, les principales contributions ainsi que sa validation et sa valorisation au travers de publications. À la

[1] Mes autres travaux sont brièvement présentés dans mon curriculum vitae détaillé.

suite de quoi, je pose un bilan avant d'ouvrir différentes voies au travers de perspectives qui représentent mon projet scientifique. Je l'ai intitulé « *vers une industrialisation de l'évolution logicielle* » afin de prendre en compte le contexte des nouvelles générations d'architectures logicielles. L'objectif est de tendre vers des systèmes logiciels durables dans le temps, en augmentant leur espérance de vie, idéalement de manière continue. J'attache également à ce projet des pistes qui me paraissent d'importance : la formation universitaire et les collaborations avec le monde socio-économique.

Avant-propos

Le présent document ne prétend pas présenter l'ensemble des travaux de recherche que j'ai menés depuis ma prise de fonction à l'Université de Nantes en 2001. Il se focalise sur les travaux dédiés aux problématiques d'évolution des architectures logicielles à composants et à services. Ces travaux ont comme socle commun de s'adresser à une même catégorie de problèmes et à une même catégorie d'utilisateurs : appréhender la complexité des architectures logicielles durant leur phases de conception et leur cycle de vie en proposant une assistance aux architectes logiciels. Plus précisément, les différents besoins abordés sont d'aider à faire face à : l'évolution, la co-évolution, la maintenance, la rétro-conception, la migration, la restructuration, la mémoire du savoir-faire d'évolution. L'approche récurrente est de spécifier et réutiliser toute connaissance jugée utile, qu'elle soit statique ou dynamique.

La Figure 1 présente l'ensemble de mes travaux depuis que je suis en poste. Ils sont déclinés selon les paradigmes et les thèmes scientifiques abordés (les deux premières lignes). L'axe des abscisses présente la chronologie. L'axe des ordonnées présente les différents types de travaux menés (post-doctorant, thèses, master…) dans le temps et par paradigme/thème scientifique. Seuls les travaux principaux, encadrés (en bleu), seront détaillés dans le présent document.

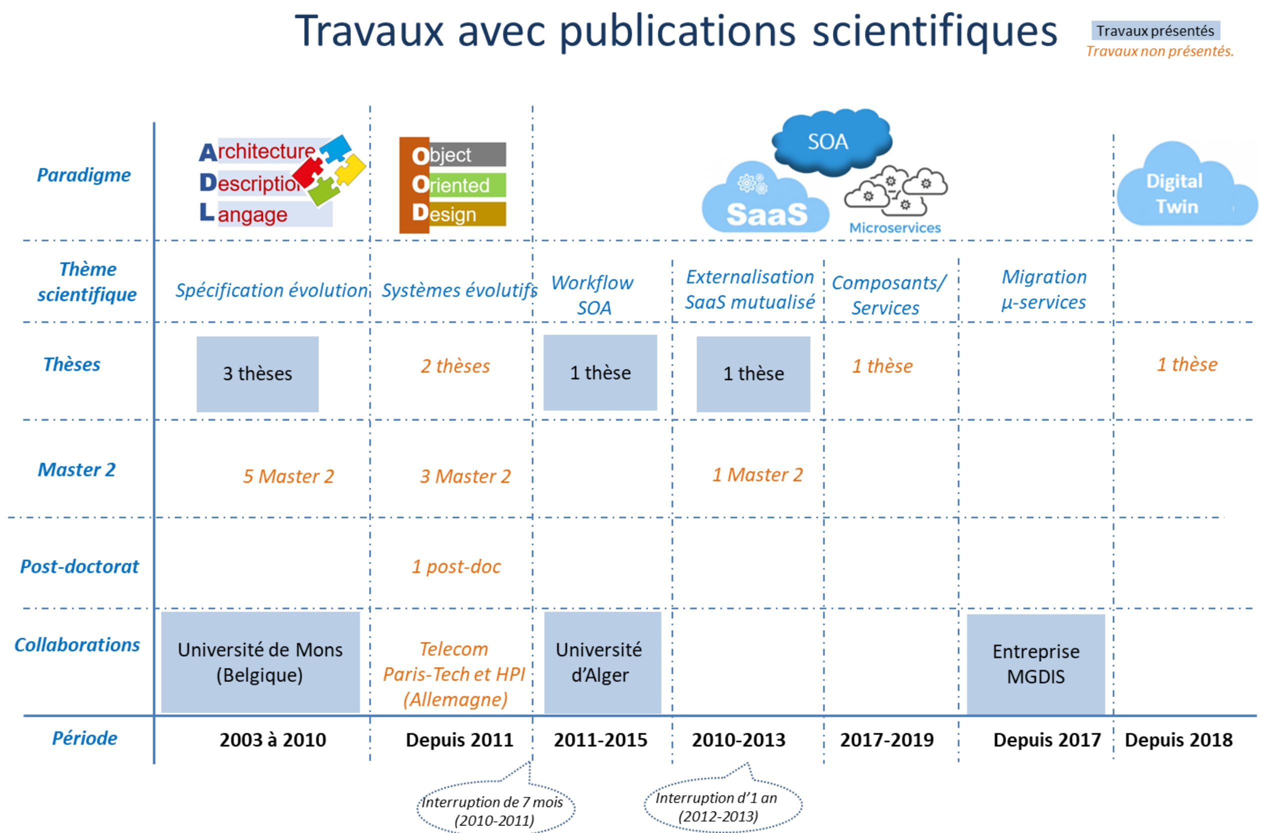


Figure 1: Trame générale des travaux avec publications scientifiques.

L'ensemble de ces travaux ont été menés avec différents collaborateurs dont mes doctorantes et doctorants. A l'apport de mes collaborateurs, s'ajoute celui de mon double profil universitaire Génie Logiciel et Systèmes d'Information : je considère les problématiques d'évolution logicielle à un haut niveau d'abstraction. L'évolution logicielle induit souvent des coûts faramineux et des effets de bord

désastreux. Pour mieux la maîtriser et la contrôler, mon objectif est d'arriver à rendre les architectures logicielles intrinsèquement évolutives et à architecturer l'évolution logicielle. Pour cela, le fil directeur de mes travaux a été de faire une *séparation claire des préoccupations* entre les aspects fonctionnels des architectures logicielles de leurs aspects évolutifs. Cela implique donc de considérer l'évolution logicielle comme un système à part entière à étudier, à identifier ses parties constituantes et à les structurer. Cela revient à œuvrer pour une ingénierie de l'évolution logicielle et non à la traiter de manière ad hoc.

Mes travaux, depuis que je suis en poste, ont été de 2002 à 2010 académiques. A partir de 2010, à la suite d'une année de CRCT[2] à l'étranger (Belgique, Angleterre, Etats-Unis) ainsi qu'au détour d'une thèse CIFRE, mes travaux se sont imprégnés d'un contexte de terrain en lien avec le monde économique et continuent de l'être. J'apprécie les deux types de travaux, académiques et ceux en lien avec le monde socio-économique, ainsi que leur cohabitation. Ils s'enrichissent mutuellement.

[2] Congés pour Recherche ou Conversion Thématique

SOMMAIRE

Table des matières 1 niveau

Table des matières 2 niveaux

Résumé 4
Avant-propos 6
Chapitre 1. Introduction 16
1. Complexité logicielle croissante 16
2. Mon identité scientifique 18
3. Organisation générale du document 19
Chapitre 2. Contexte : évolution et architectures logicielles 20
1. L'évolution logicielle : un vaste problème 20
2. Les architectures logicielles : définitions fondatrices 24
3. Architecturer l'évolution 26
Chapitre 3. Architectures logicielles à base de composants : contributions ... 30
1. Introduction 31
2. Un modèle pour l'Evolution Structurelle dans les Architectures Logicielles ... 35
3. Capitalisation et réutilisation d'évolutions : styles d'évolution 45
4. Extraction d'architectures logicielles d'un système orienté objet : une approche par exploration ... 55
5. Spécification de processus de migrations par approche formelle : théorie des graphes .. 66
6. Conclusion 74
Chapitre 4. Architectures logicielles à services (SOA, SaaS mutualisé, micro-services) : contributions ... 75
1. Introduction 76
2. Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA ... 85
3. Externalisation du modèle SaaS mutualisé et de sa gestion 96
4. Capitalisation de migrations vers des architectures micro-services 106
5. Conclusion 113

Chapitre 5. Conclusion et Perspectives 114
1. Introduction 114
2. Bilan 115
3. Perspectives scientifiques : vers une industrialisation du Génie Logiciel 117
4. Perspectives vers la formation universitaire et le monde socio-économique 123
Chapitre 6. Bibliographie 125
Références citées 125

Table des matières 3 niveaux

Résumé 4
Avant-propos 6
Chapitre 1. Introduction 16
1. Complexité logicielle croissante 16
2. Mon identité scientifique 18
3. Organisation générale du document 19
Chapitre 2. Contexte : évolution et architectures logicielles 20
1. L'évolution logicielle : un vaste problème 20
1.1. L'évolution logicielle sur le plan économique 21
1.2. L'évolution logicielle sur le plan scientifique 22
2. Les architectures logicielles : définitions fondatrices 24
3. Architecturer l'évolution 26
3.1. Constat sur l'évolution des architectures logicielles 26
3.2. Approche générale préconisée 27
3.3. Enjeux : systèmes évolutifs et supports explicites à l'évolution 27
3.4. Fondements et positionnement de l' approche 28
3.5. Trame de présentation de mes travaux 28
Chapitre 3. Architectures logicielles à base de composants : contributions 30
1. Introduction 31
1.1. Architectures logicielles à base de composants et leurs langages de description 31
1.2. Concepts fondamentaux 32
1.3. Survol des travaux d'évolution structurelle d'architectures logicielles à base de composants.. 33
2. Un modèle pour l'Evolution Structurelle dans les Architectures Logicielles 35
2.1. Fiche d'identité 35
2.2. Synthèse 35
2.3. Verrous scientifiques 36

2.4. SAEV : un modèle d'évolution d'architectures logicielles 36
2.5. Propriétés sémantiques des connecteurs 41
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) 43
2.7. Brève synthèse des travaux fondateurs 43
2.8. Positionnement et bilan 43

3. Capitalisation et réutilisation d'évolutions : styles d'évolution 45
3.1. Fiche d'identité 45
3.2. Synthèse 45
3.3. Verrous scientifiques 46
3.4. SAEM : Style-based Architectural Evolution Model 46
3.5. Bibliothèques pour les styles d'évolution 50
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) 52
3.7. Brève synthèse des travaux fondateurs 53
3.8. Positionnement et bilan 53

4. Extraction d'architectures logicielles d'un système orienté objet : une approche par exploration 55
4.1. Fiche d'identité 55
4.2. Synthèse 55
4.3. Verrous scientifiques 56
4.4. La démarche ROMANTIC pour l'extraction. 56
4.5. Mise en pratique 62
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) 64
4.7. Brève synthèse des travaux fondateurs 64
4.8. Positionnement et bilan 65

5. Spécification de processus de migrations par approche formelle : théorie des graphes 66
5.1. Fiche d'identité 66
5.2. Synthèse 66
5.3. Verrous scientifiques 67
5.4. Concepts et opérations pour l'évolution 67
5.5. Le patron d'évolution 69
5.6. Contexte des travaux (thèses, M2, contrat, collaborations) 73
5.7. Brève synthèse des travaux fondateurs 73
5.8. Positionnement et bilan 74

6. Conclusion 74

Chapitre 4. Architectures logicielles à services (SOA, SaaS mutualisé, micro-services) : contributions 75

1. Introduction 76
1.1. Architectures Logicielles et notion de Service 76
1.2. Restructuration, externalisation, migration autour de la notion de *service* 84

2. Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA 85
2.1. Fiche d'identité 85
2.2. Synthèse 85
2.3. Verrous scientifiques 87

2.4. Interconnexion de WF avec les Patrons de coopération à base de services 87
2.5. Support de la flexibilité des modèles de WFIO 92
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) 93
2.7. Brève synthèse des travaux fondateurs 93
2.8. Positionnement et bilan 94

3. Externalisation du modèle SaaS mutualisé et de sa gestion 96
3.1. Fiche d'identité 96
3.2. Synthèse 96
3.3. Verrous scientifiques 97
3.4. Gestion de la Variabilité des besoins de locataires 97
3.5. Modélisation de la variabilité : nos contributions 100
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) 104
3.7. Brève synthèse des travaux fondateurs 104
3.8. Positionnement et bilan 105

4. Capitalisation de migrations vers des architectures micro-services 106
4.1. Fiche d'identité 106
4.2. Synthèse 106
4.3. Verrous scientifiques 107
4.4. Migration de monolithes vers les micro-services : recommandations techniques 108
4.5. Migration de monolithes vers les micro-services : recommandations fonctionnelles 110
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) 112
4.7. Brève synthèse des travaux fondateurs 112
4.8. Positionnement et bilan 112

5. Conclusion 113

Chapitre 5. Conclusion et Perspectives 114

1. Introduction 114

2. Bilan 115
2.1. Contributions 115
2.2. Echecs 116

3. Perspectives scientifiques : vers une industrialisation du Génie Logiciel 117
3.1. Introduction 117
3.2. Industrialiser le Génie Logiciel ? 117
3.3. Projets scientifiques 118

4. Perspectives vers la formation universitaire et le monde socio-économique 123
4.1. Valorisation de la formation universitaire par la recherche 123
4.2. Valorisation des collaborations avec le monde socio-économique 124

Chapitre 6. Bibliographie 125

Références citées 125

Table des matières 4 niveaux

Résumé ___ *4*

Avant-propos ___ *6*

Chapitre 1. Introduction ___ *16*

1. Complexité logicielle croissante ___ **16**

2. Mon identité scientifique ___ **18**

3. Organisation générale du document ___ **19**

Chapitre 2. Contexte : évolution et architectures logicielles ___ *20*

1. L'évolution logicielle : un vaste problème ___ **20**
1.1. L'évolution logicielle sur le plan économique ___ 21
1.2. L'évolution logicielle sur le plan scientifique ___ 22

2. Les architectures logicielles : définitions fondatrices ___ **24**

3. Architecturer l'évolution ___ **26**
3.1. Constat sur l'évolution des architectures logicielles ___ 26
3.2. Approche générale préconisée ___ 27
3.3. Enjeux : systèmes évolutifs et supports explicites à l'évolution ___ 27
3.4. Fondements et positionnement de l' approche ___ 28
3.5. Trame de présentation de mes travaux ___ 28

Chapitre 3. Architectures logicielles à base de composants : contributions ___ *30*

1. Introduction ___ **31**
1.1. Architectures logicielles à base de composants et leurs langages de description ___ 31
1.2. Concepts fondamentaux ___ 32
1.3. Survol des travaux d'évolution structurelle d'architectures logicielles à base de composants 33

2. Un modèle pour l'Evolution Structurelle dans les Architectures Logicielles ___ **35**
2.1. Fiche d'identité ___ 35
2.2. Synthèse ___ 35
2.3. Verrous scientifiques ___ 36
2.4. SAEV : un modèle d'évolution d'architectures logicielles ___ 36
a. Objectif ___ 36
b. Méta-modèle et concepts de SAEV ___ 37
c. Illustration des règles ___ 38
d. Processus d'évolution ___ 39
e. Conclusion ___ 40
2.5. Propriétés sémantiques des connecteurs ___ 41
a. Objectif ___ 41
b. Propriétés sémantiques ___ 41
c. Conclusion ___ 42
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) ___ 43
2.7. Brève synthèse des travaux fondateurs ___ 43
2.8. Positionnement et bilan ___ 43

3. Capitalisation et réutilisation d'évolutions : styles d'évolution ____ 45
3.1. Fiche d'identité ____ 45
3.2. Synthèse ____ 45
3.3. Verrous scientifiques ____ 46
3.4. SAEM : Style-based Architectural Evolution Model ____ 46
a. Méta-modèle SAEM et principaux concepts ____ 46
b. Représentation des styles d'évolution ____ 49
c. Conclusion ____ 50
3.5. Bibliothèques pour les styles d'évolution ____ 50
a. Niveaux de modélisation des bibliothèques ____ 50
b. Les acteurs ____ 51
c. Élaboration d'une bibliothèque de styles d'évolution ____ 52
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) ____ 52
3.7. Brève synthèse des travaux fondateurs ____ 53
3.8. Positionnement et bilan ____ 53

4. Extraction d'architectures logicielles d'un système orienté objet : une approche par exploration ____ 55
4.1. Fiche d'identité ____ 55
4.2. Synthèse ____ 55
4.3. Verrous scientifiques ____ 56
4.4. La démarche ROMANTIC pour l'extraction. ____ 56
a. Un modèle de correspondance Objet/Architecture ____ 57
b. Modèle de mise en correspondance ____ 58
c. Guider le processus d'extraction : ____ 59
d. Qualité d'une architecture ____ 60
e. Processus d'extraction des informations intentionnelles ____ 62
4.5. Mise en pratique ____ 62
a. Algorithmes pour l'exploration ____ 62
b. Cas d'étude ____ 63
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) ____ 64
4.7. Brève synthèse des travaux fondateurs ____ 64
4.8. Positionnement et bilan ____ 65

5. Spécification de processus de migrations par approche formelle : théorie des graphes 66
5.1. Fiche d'identité ____ 66
5.2. Synthèse ____ 66
5.3. Verrous scientifiques ____ 67
5.4. Concepts et opérations pour l'évolution ____ 67
5.5. Le patron d'évolution ____ 69
a. Le processus d'évolution semi-formel ____ 69
b. Validation par la théorie des graphes ____ 70
5.6. Contexte des travaux (thèses, M2, contrat, collaborations) ____ 73
5.7. Brève synthèse des travaux fondateurs ____ 73
5.8. Positionnement et bilan ____ 74

6. Conclusion ____ 74

Chapitre 4. Architectures logicielles à services (SOA, SaaS mutualisé, micro-services) : contributions _____ 75

1. Introduction _____ 76
1.1. Architectures Logicielles et notion de Service _____ 76
a. Notion de service _____ 76
b. Une implémentation de service : les services Web _____ 77
c. Architecture orientée service _____ 77
d. Le Cloud computing _____ 79
e. Le SaaS (Software-as-a-Service) et ses niveaux de maturité _____ 81
f. Micro-services _____ 82
1.2. Restructuration, externalisation, migration autour de la notion de *service* _____ 84

2. Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA _____ 85
2.1. Fiche d'identité _____ 85
2.2. Synthèse _____ 85
2.3. Verrous scientifiques _____ 87
2.4. Interconnexion de WF avec les Patrons de coopération à base de services _____ 87
a. Méta-modèles de worflow _____ 88
b. Démarche de restructuration et d'interconnexion de processus _____ 90
c. Approche à base de patrons : les PCBS _____ 90
2.5. Support de la flexibilité des modèles de WFIO _____ 92
a. Adaptabilité et adaptation d'un modèle de WFIO _____ 92
b. Evolutivité et Evolution des modèles de WFIO _____ 93
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) _____ 93
2.7. Brève synthèse des travaux fondateurs _____ 93
2.8. Positionnement et bilan _____ 94

3. Externalisation du modèle SaaS mutualisé et de sa gestion _____ 96
3.1. Fiche d'identité _____ 96
3.2. Synthèse _____ 96
3.3. Verrous scientifiques _____ 97
3.4. Gestion de la Variabilité des besoins de locataires _____ 97
a. Le style architectural du modèle SaaS mutualisé _____ 97
b. Point sur la gestion de la variabilité _____ 98
c. Constat et approche suivie _____ 99
3.5. Modélisation de la variabilité : nos contributions _____ 100
a. Notre méta-modèle de variabilité : concepts et formalisations _____ 100
b. Variabilité sous forme d'un service : l'architecture de VaaS et le processus d'externalisation de la gestion de la variabilité _____ 102
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) _____ 104
3.7. Brève synthèse des travaux fondateurs _____ 104
3.8. Positionnement et bilan _____ 105

4. Capitalisation de migrations vers des architectures micro-services _____ 106
4.1. Fiche d'identité _____ 106
4.2. Synthèse _____ 106
4.3. Verrous scientifiques _____ 107

4.4. Migration de monolithes vers les micro-services : recommandations techniques 108
a. Granularité des services 108
b. Déploiement des services 109
c. Intégration de services 109
d. Bilan 110
4.5. Migration de monolithes vers les micro-services : recommandations fonctionnelles 110
a. Suivre une segmentation fonctionnelle 110
b. Adopter une approche de standards/normalisation : 111
c. A propos de la granularité de micro-services : 112
d. Importance de la sémantique 112
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) 112
4.7. Brève synthèse des travaux fondateurs 112
4.8. Positionnement et bilan 112

5. Conclusion 113

Chapitre 5. Conclusion et Perspectives 114

1. Introduction 114

2. Bilan 115
2.1. Contributions 115
a. Sur les architectures logicielles à base de composants (2002 – 2010) 115
b. Sur les architectures logicielles autour du service (depuis 2010) 116
2.2. Echecs 116

3. Perspectives scientifiques : vers une industrialisation du Génie Logiciel 117
3.1. Introduction 117
3.2. Industrialiser le Génie Logiciel ? 117
3.3. Projets scientifiques 118
a. Vers une adoption aisée du SaaS mutualisé 118
b. Architecture logicielle générique pour le Jumeau Numérique 119
c. Vers la prise en compte de la stratégie d'entreprise 121
d. Architecture logicielles continues et aide à la décision dans un environnement complexe et incertain 123
e. Collaborations scientifiques 123

4. Perspectives vers la formation universitaire et le monde socio-économique 123
4.1. Valorisation de la formation universitaire par la recherche 123
4.2. Valorisation des collaborations avec le monde socio-économique 124

Chapitre 6. Bibliographie 125

Références citées 125

Chapitre 1.

Introduction

Sommaire

1. Complexité logicielle croissante ______ 16
2. Mon identité scientifique ______ 18
3. Organisation générale du document ______ 19

1. Complexité logicielle croissante

Le constat clé est que plus un système logiciel devient complexe, plus il devient nécessaire d'expliciter ses différentes facettes. Face à l'accroissement de la complexité, la vision de l'ingénierie des logiciels s'est alors radicalement modifiée en passant de la perspective de développement d'un système monolithique – ou « monobloc » – vers un système construit comme un assemblage de « morceaux » logiciels. Ainsi, des catégories d'architectures ont émergées, offrant chacune des unités granulaires différentes et introduisant des niveaux d'abstractions pour le partitionnement d'un système (confère Figure 1). Le besoin de notations adéquates pour la construction à large échelle est également apparu.

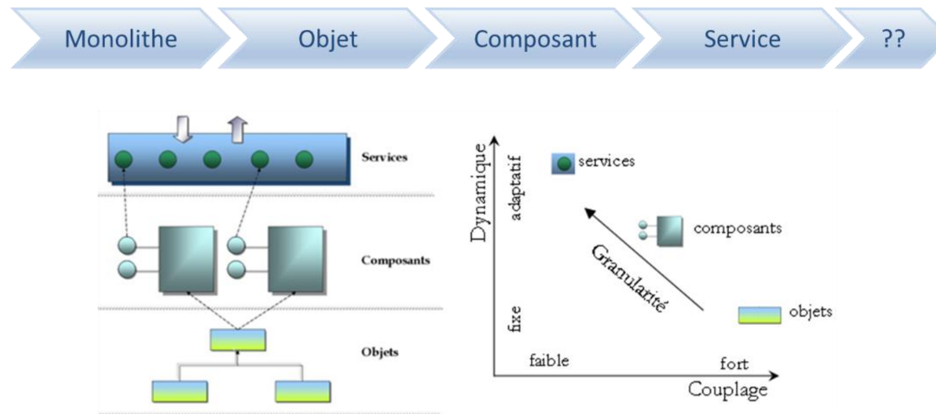


Figure 2 : Catégories d'architectures logicielles[3].

Le numéro spécial de l'été 2006 de l'IEEE Software magazine a été dédié au domaine de l'architecture logicielle pour commémorer deux décennies de recherche et une décennie de pratique [Shaw& 06] [Kruchten& 06]. L'architecture logicielle, depuis son avènement dans les années 90 [Perry& 92], [Shaw& 96], [Bass& 03], est devenue importante pour les systèmes logiciels car elle est au cœur du processus de prise de décision de la conception architecturale d'un système logiciel. Elle est le lieu de la spécification de toutes les décisions de conception architecturale. Elle fournit également une abstraction réutilisable d'un système logiciel. Cette abstraction, transférable à d'autres systèmes ayant des exigences similaires, favorise ainsi la réutilisation à grande échelle car la discipline de l'architecture logicielle concerne essentiellement les systèmes complexes et à large échelle.

Une « bonne » architecture a une influence positive sur la qualité du système final alors qu'une « mauvaise » architecture peut avoir des conséquences désastreuses, jusqu'à l'arrêt du projet [Garlan 00]. L'intégration de l'architecture logicielle dans le cycle de développement des logiciels est jugée indispensable sous peine de faire échouer les projets :

> *« Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled. »*
>
> Eoin Woods [Bass& 03]

Je considère que le rôle de l'architecture logicielle dans le cycle de vie du système logiciel opérationnel est aussi important que son rôle dans son cycle de développement. Comme tout système logiciel, une propriété intrinsèque d'une architecture logicielle est son besoin d'évoluer, sans pour autant avoir les aptitudes appropriées à le faire. Elle connait des problématiques d'évolution encore plus critiques que les modèles de conception et le code car elle est le siège des décisions architecturales stratégiques et elle impacte ainsi fortement la qualité du système logiciel final. Répondre à ces problématiques est un enjeu majeur. Fred Brooks, dans son livre 'The Mythical Man-Month: Essays on Software Engineering', indique ainsi que plus de 90% des coûts d'un système logiciel classique surviennent durant la phase de maintenance, et que toute partie réussie d'un logiciel est inévitablement appelée à être mise à jour [Brooks 95].

[3] Figure en partie tirée de http://muse-component.org/index.php?page=paradigm

2. Mon identité scientifique

Dans le film « *Jeu de Guerre* »[4], je suis interpellée par cette question posée sur un ton horrifié par un agents secret à son confrère : « *Tu sais combien ça coute de reprogrammer un satellite* ? ». Excellente question ! Une rapide recherche m'indique que fabriquer des satellites capables de s'auto-reconfigurer permettraient d'économiser des millions de dollars et de réduire la quantité de débris spatiaux en orbite. L'engin spatial pourrait même être loué à différents groupes de chercheurs au cours d'une même mission, ce qui en répartirait le coût [Sauser 09]. Voici un exemple concret et parlant quant à l'importance de réfléchir en amont aux besoins d'évolution des logiciels !

Ma conviction est l'importance d'architecturer les systèmes logiciels pour les rendre capables d'évoluer par eux-mêmes ou pour qu'ils puissent être aisément modifiés sans pour autant devoir les mettre à l'arrêt ou les couper de leur environnement. Pourquoi ? pour assurer la durabilité des systèmes logiciels en augmentant leur longévité, faisant ainsi baisser leurs coûts tout en tentant de préserver leur qualité. Comment ? Je suis profondément convaincue que la solution fondamentale réside dans le principe de la *séparation des préoccupations.*

Ossher et Tarr définissent une préoccupation [Ossher& 95] (*concern*) comme : "*A concern is the part of a software system relevant to a particular concept, goal, or purpose*". Ce principe, identifié dès 1972 par D. Parnas [Parnas 72] et aussi connu sous l'expression « diviser pour régner ». Ce principe a été fondateur pour rendre modulaire le code. L'objectif est de découper un logiciel en unités réutilisables aussi petites que possible pour mieux maîtriser la complexité de l'ensemble. Je prône cette approche à toutes les étapes du cycle de développement logiciel, à commencer par les exigences et bien sûr jusqu'au code. La principale difficulté est d'identifier cette notion de *préoccupation*. Elle est en effet particulièrement subjective et peut varier en fonction des différentes perceptions (du domaine, de l'analyste, du décideur métier, de l'architecte logiciel, du concepteur, du développeur…). La seconde difficulté est une granularité des préoccupations qui n'est pas uniforme. Les applications étant intrinsèquement évolutives, elles peuvent rencontrer de nouvelles préoccupations, connaître la modification de certaines et voir quelques-unes disparaitre.

Pour ma part, je suis convaincue que l'évolution logicielle doit être une préoccupation séparée des autres préoccupations des architectures logicielles : l'évolution logicielle est une préoccupation à part entière. Mes travaux de recherche prônent une *ingénierie de l'évolution logicielle*. Elle doit être, à ce titre, *architecturée*. Selon le Larousse, architecturer revient à « *élaborer, structurer, construire une œuvre, un ouvrage en organisant avec rigueur ses différentes parties* ». Pour cela, je m'appuie sur une approche architecturale et de méta-modélisation, pour appliquer le principe de séparation de préoccupations. Au même titre que les architectures et les modèles représentent la pierre angulaire de l'abstraction de systèmes logiciels complexes, ils représentent également un moyen approprié de gérer l'évolution des systèmes logiciels. L'objectif est de guider l'évolution logicielle, renforcer et réduire les risques critiques et les ressources importantes (par exemple, les coûts, le temps, le personnel) impliqués dans l'évolution logicielle.

Mes travaux se focalisent sur l'évolution des architectures logicielle à un haut niveau d'abstraction et considèrent différents paradigmes logiciels (orienté objet, architectures logicielles à base de composants, architectures orientées service, architectures en mode SaaS mutualisé, architectures micro-services et jumeaux numériques dans l'industrie). J'ai fait le choix de présenter une sélection

[4] 1992, réalisé par Phillip Noyce.

d'entre eux : ceux qui s'intéressent aux besoins d'évolution des architectures logicielles à base de composants et des architectures logicielles à base de services.

3. Organisation générale du document

L'organisation générale du document est schématisée en Figure 3 :

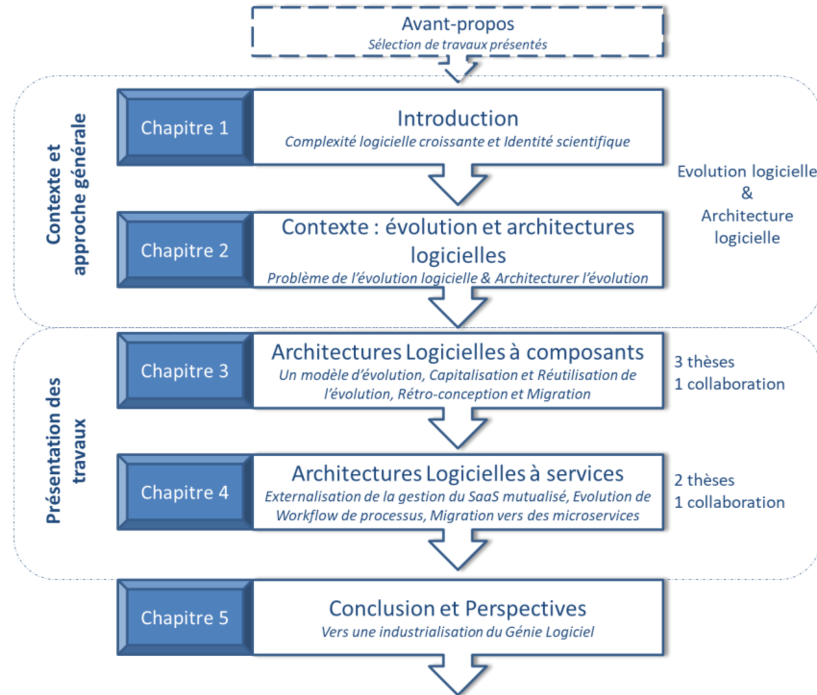


Figure 3 : Organisation schématique du document.

Après l'avant-propos, ce Chapitre 1. présente le contexte général et mon identité scientifique. Le Chapitre 2. pose le contexte de mes travaux, à savoir leurs deux socles : la problématique de l'évolution logicielle et les architectures logicielles avant de présenter le principe général de mon approche : « architecturer l'évolution ». Viennent ensuite les deux chapitres regroupant les travaux que j'ai menés avec mes collaborateurs. Le Chapitre 3. présente nos travaux et contributions sur l'évolution dans le contexte des architectures logicielles à base de composants. Il s'agit des travaux liés à la thèse de Nassima Sadou sur un modèle d'évolution ; à la thèse d'Olivier Le Goaer sur la capitalisation et réutilisation d'un savoir-faire d'évolution ; à la thèse de Sylvain Chardigny sur l'extraction de l'architecture logicielle de systèmes à objets existants et, enfin, à la collaboration que j'ai eu avec Tom Mens de l'Université de Mons (Belgique) sur la proposition de patrons d'évolution automatisés et réutilisables. Le Chapitre 4. présente nos travaux et contributions sur l'évolution dans le contexte des architectures logicielles autour de la notion de service. Il s'agit de travaux liés à la thèse de Souad Boukhedouma sur l'évolution de workflow de processus avec une approche SOA ; à la thèse de Ali Ghaddar sur l'externalisation de la gestion de la variabilité d'applications SaaS mutualisées, et enfin, à la collaboration avec Jean-Philippe Gouigoux, directeur technique de l'éditeur logiciel MGDIS, basé à Vannes, sur des retours d'expériences de migration de systèmes monolithes vers des architectures micro-services. Le Chapitre 5. conclut ce document avec un bilan et des perspectives scientifiques sur les années à venir ainsi que des réflexions sur leurs potentiels impacts sur la formation universitaires et les collaborations avec le monde socio-économiques.

Chapitre 2.

Contexte : évolution et architectures logicielles

SOMMAIRE

1. L'évolution logicielle : un vaste problème ___ 20
1.1. L'évolution logicielle sur le plan économique ___ 21
1.2. L'évolution logicielle sur le plan scientifique ___ 22

2. Les architectures logicielles : définitions fondatrices ___ 24

3. Architecturer l'évolution ___ 26
3.1. Constat sur l'évolution des architectures logicielles ___ 26
3.2. Approche générale préconisée ___ 27
3.3. Enjeux : systèmes évolutifs et supports explicites à l'évolution ___ 27
3.4. Fondements et positionnement de l' approche ___ 28
3.5. Trame de présentation de mes travaux ___ 28

Ce chapitre présente les deux principaux axes dans lesquels s'inscrivent mes travaux de recherche, l'évolution logicielle et les architectures logicielles, ainsi que l'approche générale de mes travaux qui a pour objectifs d'*architecturer l'évolution logicielle*.

1. L'évolution logicielle : un vaste problème

L'évolution de systèmes complexes est une activité à hauts risques. Les changements rencontrés vont des changements techniques (dus à l'évolution rapide des plateformes technologiques) aux modifications des applications elles-mêmes (dues à l'évolution naturelle des activités soutenues par ces applications logicielles). Ces changements conduisent à un besoin croissant de techniques disciplinées et d'outils d'ingénierie pour soutenir un large éventail d'activités de développement.

Différents types de changements existent en Génie Logiciel : *évolution*, *maintenance*, *adaptation*, *migration*, *rétro-conception*, *corrections*, *restructuration*, *versionnement*…. La *maintenance* est souvent définie comme la totalité des activités requises pour fournir un support, à un coût rentable pour un système logiciel, réalisées de l'étape de pré-livraison à la post-livraison [Pigoski 96]

[Bennett& 00] [Chapin& 01] [Anquetil& 07]. L'*adaptation*, quant à elle, a pour objectif d'assurer le bon fonctionnement d'un système confronté à un changement de contexte [Canal& 06]. La *migration*, selon l'ANSI/IEEE Std 729-1983 est la « *modification d'un produit logiciel après livraison pour corriger des défauts, améliorer les performances ou d'autres attributs, ou pour adapter le produit à un environnement modifié* ». Elle représente donc le processus qui consiste à mettre à niveau ou à remplacer tout ou partie d'un système logiciel [Fleurey& 07], [Varma& 07]. Le *reverse engineering* (ou *rétro*-conception) est le processus d'analyse d'un système (souvent existant mais pas seulement) pour identifier ses composantes et leurs interrelations et les abstraire de leur mise en œuvre. Il s'agit de créer des représentations du système sous une autre forme ou à un niveau plus élevé d'abstraction [Chikofsky&, 90] [Canfora& 11] [Muller& 00]. Il est à noter qu'en soi le reverse-engineering n'implique pas de changer le système. Il permet d'en changer indirectement la perception. La *réingénierie* (ou *rénovation*) est l'examen et la modification d'un système pour le reconstruire sous une nouvelle forme. La réingénierie inclut généralement une phase de reverse-engineering pour obtenir une abstraction décorrélée des détails techniques, suivie d'une forme d'ingénierie avancée ou de restructuration [Chikofsky&, 90]. D'autres changements existent également tels que la *restructuration* (transformation d'une représentation à une autre au même niveau d'abstraction, plus connu en objet par le *refactoring* [Mens& 04]), le *forward engineering* (passer de la conception abstraite à la mise en œuvre physique), la *redocumentation* ou encore le *design recovery* [Chikofsky& 90].

Je considère que le terme *évolution* englobe l'ensemble des changements et des transformations d'un système logiciel tout au long de son cycle de vie. Cette déclinaison de types de changements et les solutions existantes par types mettent en évidence que la problématique de l'évolution logicielle est complexe. Elle l'est autant sur le plan économique que sur le plan scientifique :

1.1. L'évolution logicielle sur le plan économique

Malgré son importance majeure pour l'industrie et le rappel constant de la communauté scientifique, maintenir les logiciels, et notamment la cohérence entre les artefacts logiciels appelé à évoluer pour répondre aux exigences et aux contraintes multiples et variées, est traitée après coup [Finkelstein& 94], [Bain 08], [Tröls& 19]. Plus de quarante ans après l'apparition des fameuses lois de Lehman sur l'évolution logicielle [Lehman 78], quel est le chemin parcouru ? Les préoccupations d'évolution vont du plus bas niveau de code ou d'infrastructures aux plus hautes considérations stratégiques (Figure 1). Les coûts induits par la maintenance et l'évolution de logiciels existants restent particulièrement élevés.

La gestion de l'évolution logicielle présente un caractère crucial dans les contextes actuels, particulièrement mouvants, et les environnements multi-utilisateurs. Les évolutions et l'usage intensif des technologies de l'information et notamment de l'Internet et de la démocratisation des produits connectés (ordinateurs portables, tablettes, smartphones, montres, IoT…) ont énormément complexifié les informations et leur circulation, qui sont de plus en plus réparties et décloisonnées. Ces dernières années, les avancées dans le numérique, ont permis aux organisations de se tourner vers leur environnement, permettant automatiquement à des informations clés de provenir aussi bien de l'intérieur que de l'extérieur, et d'être exploitées. Ces informations d'un incroyable volume, de différents formats, provenant d'une multitude de canaux et dans des environnements dynamiques et changeants, devront être absorbées et interprétées pour le fonctionnement mais également pour l'innovation (supervision numérique, meilleure efficacité, augmenter la compétitivité, augmenter la productivité, réduire les coûts…), ouvrant ainsi vers de nouveaux modèles économiques.

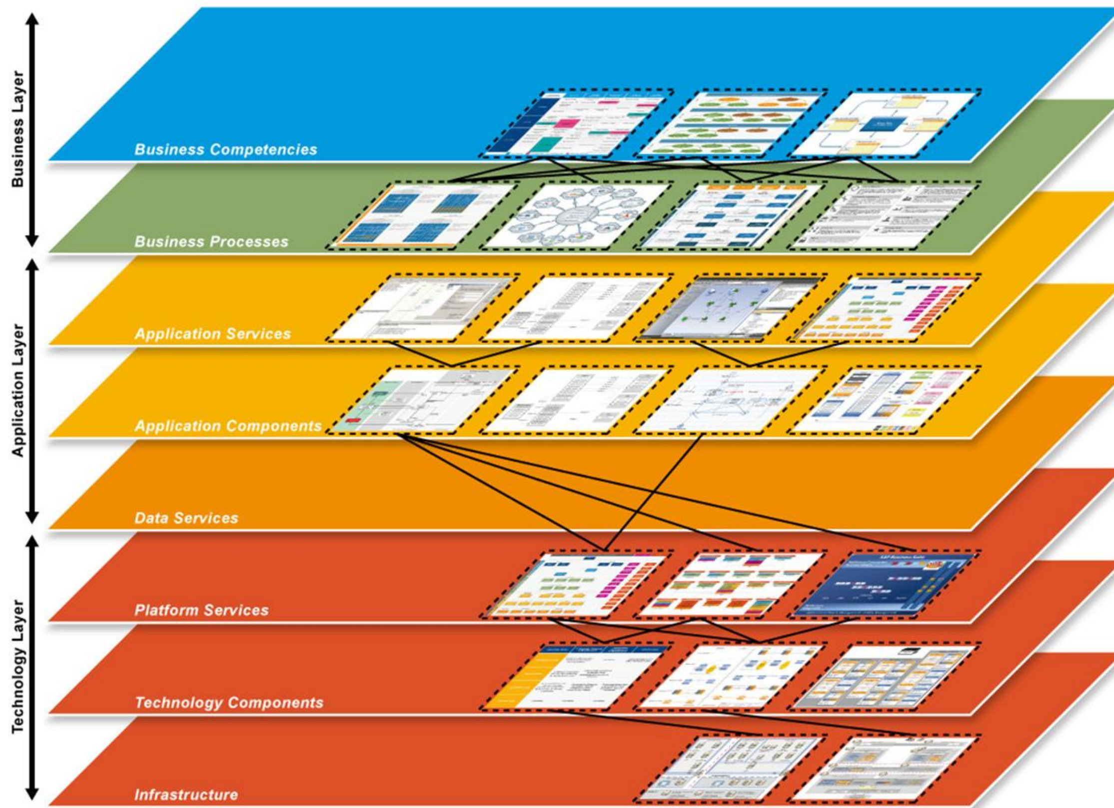


Figure 4 : architectures en couches en génie logiciel (Sysco Labs 2014)[5].

Dans ce contexte, faire évoluer un système logiciel complexe est une activité à haut risque et requiert des efforts considérables. Les chiffres sont vertigineux : maintenir un système existant revient 2 à 100 fois plus cher que d'en développer un nouveau [Sommerville 07][Fenton& 14]. Les coûts, en termes financiers, de personnels et de temps, augmentent avec l'ancienneté et la vétusté des systèmes. Leur évolution provoque le plus souvent un effet de bord non souhaitable : la corruption et la détérioration de la qualité de ces systèmes, rendant leur évolution encore plus ardue.

1.2. L'évolution logicielle sur le plan scientifique

'For at least thirteen years, I have practiced a form of software system development that I would now call 'evolutionary', but for which I had no name to begin with'. Ainsi s'exprimait Tom Gilb en 1981 [Gilb 81]. Depuis les années 80 [Lehman 78, Lehman& 85], l'évolution logicielle a été l'objet de plusieurs travaux et est devenu un domaine de recherche à part entière.

Différents travaux ont proposé des analyses et des catégorisations des approches dédiées à l'évolution logicielle. La littérature est particulièrement prolifique sur le sujet [Lehman 96], [Bennett& 00], [Mens& 01], [Lehman 05], [Madhavji& 06], [Mens& 08] et tant d'autres encore. Pour ma part, au vu de mes centres d'intérêt, je présente brièvement deux grandes catégories d'approches traitant de l'évolution en génie logiciel :

[5] Source : Mark Von Rosing et Maria Hove

1. *Support pour l'évolution logicielle au niveau du code :* la préoccupation de la maintenance du code logiciel a été présente dès les débuts du génie logiciel, d'où les grands acquis de la modularisation du code [Parnas 72], les indicateurs tels que le couplage et la cohésion [Arisholm & 04] [Bieman& 95], le refactoring du code [Baldwin& 99], et bien d'autres [Ghezzi& 91]. Cela a permis la modularisation et l'encapsulation, l'analyse de la compatibilité et de la substituabilité des modules [Chaki& 04], et les modèles de conception qui soutiennent la maintenabilité [Gamma& 94]. Cependant, le plus souvent, les opérations d'évolution, telles que l'ajout, la suppression ou la modification, sont des opérations définies en dur et « noyées » dans les spécifications du système logiciel. Il est quasiment impossible de les modifier et de les faire évoluer, du fait qu'elles ne soient pas réifiées. Il s'agit d'un *support implicite* à l'évolution. Cette situation se retrouve dans les langages de programmation objets classiques (C, Ada, Java, C++, .NET…). Ces travaux ne traitent pas d'une réorganisation à grande échelle fondée sur des abstractions architecturales. En effet,

2. *Support pour l'évolution logicielle au niveau de l'architecture :* la préoccupation est de capturer les structures essentielles de haut niveau qui sont nécessaires pour raisonner sur l'architecture d'un système logiciel complexe. Les opérations d'évolution peuvent être un ensemble d'opérations explicites et prédéfinies mais immuables, comme cela est le cas dans les premiers langages de description d'architectures (ADL) (Darwin [Magee& 95], Wright [Allen& 96], C2SADEL [Medvidovic& 99], C2 [Medvidovic & 96]). Le support à l'évolution est ici *prédéfini*. Mes travaux s'inscrivent dans cette catégorie.

Outre les indicateurs de publications scientifiques, les activités de recherche dédiées à l'évolution logicielle se reflètent dans les projets, actions et groupes de recherche. Parmi les actions menées, des projets de grande envergure ont vu le jour (ADM par l'OMG et son modèle KDM[6] dédié à la mise en place d'un consensus pour la modernisation de systèmes existants, CMMI et son modèle S3M[7] d'évaluation de la capacité de la maintenance du logiciel, la norme ISO14764 de la maintenance du logiciel…). A ces projets, s'ajoutent et se multiplient les groupes de travail et de recherche. Parmi eux : Gaudí System Architecting homepage (http://www.gaudisite.nl/) qui s'intéresse à l'architecture des systèmes, l'université de Californie San Diego (http://www-cse.ucsd.edu/~wgg/swevolution.html) qui s'intéresse à l'ubiquité et aux techniques automatique pour aider à la maintenance, la compréhension et la restructuration de très large systèmes logiciels et le groupe français RIMEL du GDR CNRS GPL qui s'intéresse à la rétro ingénierie, la maintenance et l'évolution logicielle (dont je suis membre). En France, parmi les travaux notables sur le sujet de l'évolution logicielle, ceux de Stéphane Ducasse et son équipe INRIA RMoD, notamment sur les aspects reengineering, software analysis, program visualization, reverse engineering, Lionel Seinturier et son équipe Spirals, du laboratoire CRIStAL à Lille, qui s'intèresse aux problèmes d'auto-adaptation pour les services distribués et les grands systèmes logiciels, l'équipe MAREL du LIRMM qui s'intéresse notamment à l'ingénierie de la réutilisation et de la variabilité ou encore l'équipe ArchWare de Flavio Oquendo à l'IRISA qui travaille entre autres sur la problématique d'évolution de systèmes-de-systèmes.

Aux projets et groupes de travail, s'ajoutent les manifestations scientifiques. Les conférences de premier plan telles que OOPSLA, ECOOP, ICSE, ASE, FSE/FSEC ou encore VLDB dédient des sessions à l'évolution et accordent un espace privilégié à des ateliers dédiés à l'évolution, la

6 http://www.omg.org/technology/documents/modernization_spec_catalog.htm
7 http://www.s3m.ca/fr/services/certification.html

maintenance, la réingénierie ainsi qu'à des conférences invitées dédiées au sujet. A ces manifestations, s'ajoutent les conférences et ateliers entièrement dédiés telles que ICSM, CSMR ou encore IRI ou l'atelier international Models and Evolution, dont je suis co-fondatrice. Des revues internationales publient régulièrement des articles traitant de l'évolution et dont certaines sont entièrement dédiées à la thématique tel *que Journal of Software Maintenance and Evolution: Research and Practice*.

2. Les architectures logicielles : définitions fondatrices

Cette discipline est centrée sur l'idée de réduire la complexité à travers l'abstraction et la séparation des préoccupations. Les travaux fondateurs des architectures logicielles ont posé des définitions fondamentales.

- *Perry et Wolf (1992) :* leur définition est restée d'actualité [Perry& 92]. Après avoir examiné les architectures dans d'autres disciplines (matériel, réseaux et bâtiments), Perry et Wolf décrivent une architecture logicielle comme « *un ensemble d'éléments architecturaux qui ont une forme particulière* ».

  Architecture = Eléments + Formes + Logique

  Les éléments sont divisés en trois catégories : éléments de traitement, éléments de donnée, et les éléments de connexion. La forme de l'architecture est donnée en énumérant les propriétés des différents éléments et les relations entre ces éléments. La logique (*rationale*) englobe, entre autres, la logique fonctionnelle et les aspects de qualité.

- *Shaw et Garlan (1996)* : une définition très populaire de l'architecture logicielle a été avancée par Garlan et Shaw [Shaw& 96]. Elle est plus restrictive que la définition de Perry et Wolf. Elle ouvre la voie des architectures logicielles à base de composants et de connecteurs. Ces deux concepts fondamentaux sont la formalisation des architectures logicielles à base de composants et de connecteurs et la distinction des composants de leurs interactions. Garlan et Shaw ont ainsi proposé qu'une architecture logicielle pour un système spécifique soit représentée comme *« un ensemble de composants de calcul - ou tout simplement de composants - accompagné d'une description des interactions entre ces composants - les connecteurs* ». Elle introduit également les styles architecturaux et les langages de description d'architectures (ADL : Architecture Description Language).

- *Bass, Clements et Kazman (1998) :* dans leur définition, Bass et al. [Bass& 98] font la distinction entre l'architecture concrète d'un système (autrement dit, son implémentation) et sa description architecturale. Ils insistent sur les propriétés extérieurement visibles d'un composant : elles traduisent les hypothèses que d'autres composants peuvent faire sur ce composant (par exemple, les services qu'il fournit, les ressources requises, ses performances, ses mécanismes de synchronisation). Ce peut être un service, un module, une bibliothèque, un processus, une procédure, un objet, une application, etc. La définition proposée met également en lumière le fait qu'un système peut avoir plusieurs structures correspondantes chacune à un point de vue, donc à une finalité ou classe de problèmes précis à résoudre. En somme, comme en architecture du bâtiment, un logiciel est représenté par plusieurs plans et schémas destinés chacun à un corps de métier et dépendants de l'étape du processus de développement.

- *Norme ISO/IEC/IEEE 42010 :* la norme ISO/IEC/IEEE 42010[8] vise essentiellement à définir les exigences sur la description des systèmes, logiciels et architectures d'entreprise. La norme permet de considérer tous les points de vue possible d'un système logiciel. Elle est également au cœur des différents frameworks d'architectures plus spécifiques (AADL, ARCHIMATE, MODAF, DODAF, NAF, TRAK, TOGAF…). Il est ainsi envisageable à la fois d'unifier les différentes perceptions d'un système logiciel tout en gardant les spécificités de chacune de ces perceptions. Dans ses recommandations (Figure 5), un système (*System*) est défini comme une collection de composants organisés afin d'accomplir une fonction ou un ensemble spécifique de fonctions. Le terme 'System' englobe alors des applications individuelles, des sous-systèmes, des familles de produits, etc. A partir de cette définition, il s'en suit que tout peut être un système pourvu qu'il satisfasse certains buts, dont celui d'accomplir une ou plusieurs fonctions. Il est ainsi possible de positionner tous les aspects d'un système logiciel selon ce méta-modèle. Initialement, du fait du contexte technologique de l'époque, l'architecture logicielle abordait la conception de structures logicielles statiques. Avec les avancées technologiques majeurs (internet, Cloud, IoT…), elle doit faire face à des systèmes plus globaux, toujours actifs, connectés à Internet et aux architectures en constante évolution.

La Figure 5 est le cœur de la description d'une architecture. Afin de pouvoir spécifier des architectures logicielles (concept *Architecture* dans la Figure 5), il est nécessaire de les décrire (concept *Architecture Description* dans la Figure 5). Il faut avant cela identifier quel type de vue est appropriée aux besoins (concept *Architecture View* dans la Figure 5). Il existe différents types de vue (parfois dénommés patrons architecturaux ou styles architecturaux) : en couches, dirigées par les événements, sans serveurs et tant d'autres.

[8] ISO/IEC/IEEE 42010 :2011 (succédant à IEEE Std 1471), *Systems and software engineering — Architecture description*, the latest edition of the original IEEE Std 1471:2000, *Recommended Practice for Architectural Description of Software-intensive Systems*. http://www.iso-architecture.org/ieee-1471/

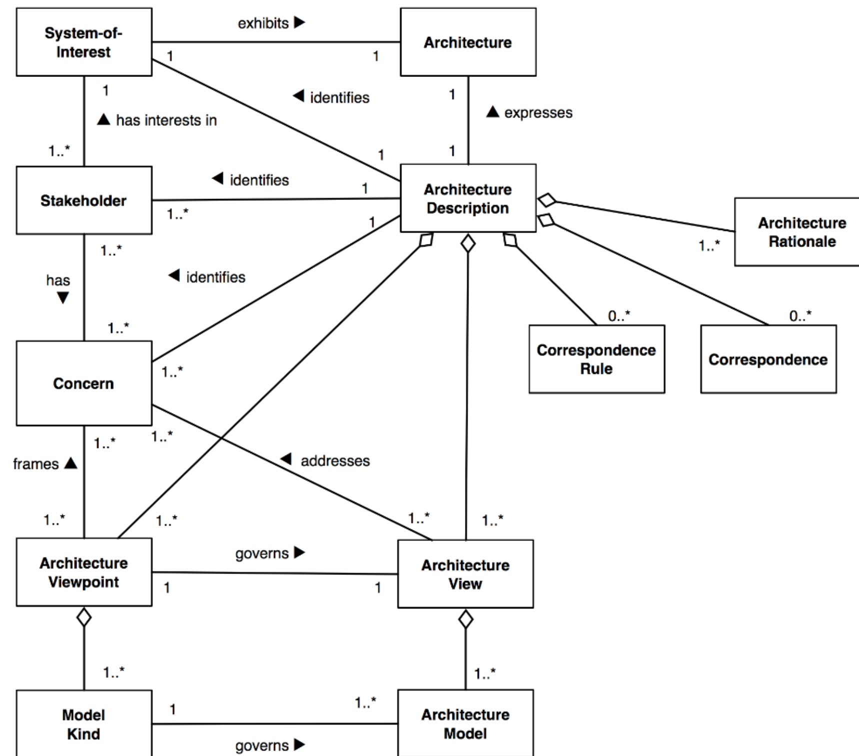


Figure 5 : Le cœur de la description de l'architecture selon ISO/IEC/IEEE 42010.

Dans le cadre de ce document, les travaux présentés concernent la vue *Composants & Connecteurs* dans le Chapitre 3. « *Architectures logicielles à base de composants* » et les vues autour de la notion de service *SOA*, *SaaS mutualisé* et *Micro-services* dans le Chapitre 4. « *Architectures logicielles à services (SOA, SaaS mutualisé, micro-services)* ».

Après avoir présenté les deux axes sur lesquels porte mes travaux, à savoir l'évolution et les architectures logicielles, je présente l'idée principale de mon approche pour architecturer l'évolution :

3. Architecturer l'évolution

Cette section présente le constat que je porte sur l'évolution des architectures logicielles, l'approche générale que je prône, les enjeux en termes de support de l'évolution, les fondements de mes travaux ainsi que la trame de leur présentation :

3.1. Constat sur l'évolution des architectures logicielles

Une architecture logicielle doit pouvoir être modifiée pour rester utilisable, réutilisable et disponible pour ses utilisateurs, et cela tout au long de son cycle de vie, voire d'allonger sa durée de vie. Le paradigme de l'architecture logicielle a favorisé la modélisation de systèmes logiciels de plus en plus complexes. Les préoccupations se sont ainsi étendues du code et des modèles vers l'architecture. Cet accroissement de la complexité s'est vu tout naturellement accompagné de questions cruciales sur l'évolution des architectures.

3.2. Approche générale préconisée

Ma vision est *d'architecturer l'évolution logicielle*. Il s'agit d'aller au-delà des solutions élaborées de manière ad hoc qui répondent à un besoin d'évolution donné et ne permettent aucune généralisation ni de réutilisation. Il s'agit de proposer des solutions de spécification de l'évolution et d'aide à l'évolution des architectures logicielles.

Il est nécessaire de considérer l'évolution comme un artefact logiciel à part entière et de l'inclure au cœur du cycle de développement dès les phases des exigences et de l'analyse [Fernández& 19]. Il faut abstraire l'évolution au travers de l'identification de concepts et processus qui lui soient propres. Cette abstraction s'exprime au sein de méta-modèles dédiés et permet également la connexion avec les architectures logicielles appelées à évoluer.

Mes travaux portent sur les deux premières strates de la couche *Application* de la Figure 4 et sur la *description d'architectures logicielles* de la Figure 5. Ma démarche s'appuie en grande partie sur des techniques et approches par abstraction, modélisation et méta-modélisation. Les solutions proposées s'adressent uniquement à des architectes logiciels experts et non à des utilisateurs finaux du système logiciel.

3.3. Enjeux : systèmes évolutifs et supports explicites à l'évolution

Je considère les systèmes logiciels selon deux catégories : les systèmes existants et les systèmes à développer « from scratch ». Les premiers font face aux problèmes de leur évolution, avec tous les coûts induits. Les seconds doivent, à mon sens, se démarquer des précédents pour ne pas rencontrer les mêmes difficultés. Ils doivent pour cela être développés en prenant en compte dès le début la dimension 'évolution' pour éviter, ou tout au moins réduire, les problèmes et les coûts prohibitifs habituellement rencontrés. Les systèmes logiciels à venir doivent être plus facilement *évolutifs*. La plupart des travaux de recherche se sont attelés à résoudre les problèmes d'évolution d'applications existantes lorsque ces problèmes sont rencontrés. Peu d'entre eux se sont focalisés sur l'explicitation de l'évolution, seul moyen à mon sens de rendre des applications aisément évolutives. Cela est principalement dû au fait que l'évolution reste encore sous-estimée dans le processus de conception et de développement logiciel. Je reste convaincue que la seule manière de dépasser et d'éviter les effets de bord du vieillissement logiciel (*software aging* introduit par D. Parnas [Parnas 94]) et du phénomène de son érosion est de placer l'évolution au cœur du processus de développement logiciel. Cette idée d'inclure l'évolution dans le cycle de vie se retrouve dans les méthodes de développement agiles [Chapin 04]. Elle représente un des défis majeurs de l'évolution logicielle [Mens& 05]. De manière complètement corrélée, l'évolution ne peut plus concerner uniquement le code d'un système mais elle doit concerner, au même titre, des artefacts de plus haut niveau d'abstraction, tels que les modèles et les architectures logicielles, mais également l'expression des besoins et leur analyse.

Je considère que l'évolution gagne à être abstraite afin d'être d'une part spécifiée et d'autre part pouvoir être gérée comme toute autre entité logicielle. Nos travaux, avec mes co-auteurs, visent à prendre en compte ces deux points :

> *Considérer et intégrer l'évolution au cœur du processus en considérant l'évolution et en la spécifiant dans des artefacts de haut niveau d'abstraction que sont les modèles.*

Cela rejoint la lignée énoncée par Mehdi Jazayeri : « *Software is more than source code; models and meta-models are important to software evolution* » [Jazayeri 05].

3.4. Fondements et positionnement de l' approche

L'approche adoptée dans mes travaux pour gérer l'évolution se résume en *abstraire* l'évolution en vue de l'*architecturer* pour offrir une *évolution explicite* de tout système appelé à évoluer :

- *Abstraire :* l'objectif de l'abstraction est de rendre accessible un élément complexe en se focalisant sur ce qui est considéré comme essentiel dans un contexte donné. Il s'agit d'extraire l'essence d'entités complexes tout en ignorant une ou plusieurs de leurs propriétés, considérées comme des détails pour un point de vue donné. Pour cela, il est nécessaire d'identifier le noyau commun de ces entités et d'en formuler des concepts généraux par extraction des caractéristiques communes à partir des entités ou d'exemples spécifiques de ces entités [Kramer 07]. L'abstraction est la clé de voûte notamment en conception. Le raisonnement par abstraction est essentiel à la construction de modèles appropriés et à leur manipulation [Alexander 96], [Gamma& 94][9]. Il passe souvent par une activité de modélisation et de méta-modélisation. *Je considère que l'essence même de l'évolution doit être identifiée et explicitée sous forme de concepts de première classe au sein d'un modèle.*
- *Architecturer :* concevoir un système nécessite de prendre en compte les différentes exigences auxquelles il est sujet. Lorsqu'un processus de conception logiciel débute, son architecte doit gérer et administrer sa construction de manière proactive, et particulièrement pour les systèmes larges et complexes [McBride 07]. Il doit en avoir une vue d'ensemble : son abstraction. *Je considère que le fait d'abstraire l'évolution au travers de concepts propres au sein de modèles permet de la considérer comme toute architecture logicielle. Elle peut par conséquent être manipulée et gérer au travers de processus et d'outils appropriés. Elle peut également évoluer.*
- *Expliciter :* grâce à l'abstraction et à la représentation sous forme d'architecture de l'évolution d'un système, il est possible de dissocier l'évolution et ses processus du comportement du système. *L'évolution devient ainsi une entité de première classe : sa spécification et sa gestion deviennent alors entièrement explicitées et explicites.*

3.5. Trame de présentation de mes travaux

La sélection des travaux présentés dans ce document se fera selon la trame suivante :

1. **Fiche synthétique :** représente la carte d'identité des travaux, mentionnant les principales informations : titre, paradigmes ciblés, les principales contributions, les mots-clés, les collaborateurs (doctorant.e.s, confrères, collègues, partenaires…), mon implication dans les travaux, le cadre de collaboration, la période ainsi que les principales publications.
2. **Synthèse :** représente un résumé de chaque travail de recherche.
3. **Verrous scientifiques :** présente les principaux verrous scientifiques traités.
4. **Contribution(s) :** une synthèse des contributions.
5. **Contexte des travaux (thèses, M2, contrat, collaborations) :** présente le contexte ayant permis de mener ces travaux ainsi que les collaborateurs.

[9] Ch. Alexander est reconnu comme l'initiateur des modèles de conception dans les années 70 et repris dans les Design Patterns du GoF.

6. **Brève synthèse des travaux fondateurs :** représente un bref état de l'art en se focalisant sur les travaux fondateurs.
7. **Positionnement et bilan :** positionne la plus-value des travaux par rapport aux principaux travaux, le bilan et mes principales contributions ainsi que les publications majeures.

Chapitre 3.

Architectures logicielles à base de composants : contributions

SOMMAIRE

1. Introduction ____ 31
1.1. Architectures logicielles à base de composants et leurs langages de description ____ 31
1.2. Concepts fondamentaux ____ 32
1.3. Survol des travaux d'évolution structurelle d'architectures logicielles à base de composants 33

2. Un modèle pour l'Evolution Structurelle dans les Architectures Logicielles ____ 35
2.1. Fiche d'identité ____ 35
2.2. Synthèse ____ 35
2.3. Verrous scientifiques ____ 36
2.4. SAEV : un modèle d'évolution d'architectures logicielles ____ 36
2.5. Propriétés sémantiques des connecteurs ____ 41
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) ____ 43
2.7. Brève synthèse des travaux fondateurs ____ 43
2.8. Positionnement et bilan ____ 43

3. Capitalisation et réutilisation d'évolutions : styles d'évolution ____ 45
3.1. Fiche d'identité ____ 45
3.2. Synthèse ____ 45
3.3. Verrous scientifiques ____ 46
3.4. SAEM : Style-based Architectural Evolution Model ____ 46
3.5. Bibliothèques pour les styles d'évolution ____ 50
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) ____ 52
3.7. Brève synthèse des travaux fondateurs ____ 53
3.8. Positionnement et bilan ____ 53

4. Extraction d'architectures logicielles d'un système orienté objet : une approche par exploration ____ 55
4.1. Fiche d'identité ____ 55
4.2. Synthèse ____ 55
4.3. Verrous scientifiques ____ 56

4.4. La démarche ROMANTIC pour l'extraction. 56
4.5. Mise en pratique 62
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) 64
4.7. Brève synthèse des travaux fondateurs 64
4.8. Positionnement et bilan 65

5. Spécification de processus de migrations par approche formelle : théorie des graphes 66
5.1. Fiche d'identité 66
5.2. Synthèse 66
5.3. Verrous scientifiques 67
5.4. Concepts et opérations pour l'évolution 67
5.5. Le patron d'évolution 69
5.6. Contexte des travaux (thèses, M2, contrat, collaborations) 73
5.7. Brève synthèse des travaux fondateurs 73
5.8. Positionnement et bilan 74

6. Conclusion 74

1. Introduction

Après avoir mené une thèse proposant un modèle d'évolution dans le paradigme objet, j'ai démarré ma carrière en recherche sur le paradigme des architectures logicielles à base de composants. Ces travaux s'adressent à la problématique de l'évolution structurelle des architectures logicielles à base de composants. Ils couvrent la période 2002-2010. Ce chapitre présente synthétiquement les travaux, tous académiques, menés avec différents collaborateurs au travers du co-encadrement de 3 thèses et une collaboration. La première thèse menée par Nassima Sadou propose un modèle pour l'Evolution Structurelle dans les Architectures Logicielles (Chapitre 3. 2.). La deuxième thèse menée par Olivier Le Goaer propose de capitaliser le savoir-faire d'évolution à des fins de réutilisation (Chapitre 3. 3.). La troisième thèse menée par Sylvain Chardigny propose une approche d'extraction de l'architecture logicielle de systèmes patrimoniaux à objets (Chapitre 3. 4.). Les derniers travaux sont issus d'une collaboration menée avec Tom Mens, Professeur à l'UMons (Belgique), qui proposent de formaliser sous forme de patron des restructurations récurrentes d'architectures logicielles (Chapitre 3. 5.).

Avant de les présenter, je rappelle les principaux concepts et notions nécessaires à la présentation des travaux :

1.1. Architectures logicielles à base de composants et leurs langages de description

Un ADL (Architecture Description Language) est un langage offrant des formalismes pour modéliser une architecture logicielle à base de composants d'un système. Un ADL fournit aussi bien une syntaxe concrète qu'un cadre conceptuel pour caractériser des architectures logicielles [Garlan& 97]. Le cadre conceptuel reflète les caractéristiques du domaine pour lequel l'ADL est prévu. Il englobe aussi leur sémantique (par exemple, les CSP [Hoare 95], les machines à états [Brand& 83]).

L'étude de Medvidovic et Tailor [Medvidovic & 00] montre la prolifération des ADLs alors qu'ils présentent beaucoup de similitudes. Néanmoins, ces ADLs ont différentes vocations, avec des préoccupations de description parfois différentes. Il existe différents ADLs, certains sont académiques et d'autres industriels. Parmi les ADLs académiques les plus connus, citons : Rapide

[Luckham& 95], UniCon [Shaw& 95], ACME [Garlan& 97], Wright [Allen& 96], C2SADEL [Medvidovic & 99], Darwin [Magee& 95], Fractal [Bruneton& 06], SafArchie [Barais 05], COSA [Smeda 06], UML2.0 [OMG], xADL2.0 [Dashofy& 01]. En dehors du cercle académique, on notera la participation des industriels aux ADLs Meta-H [Binns& 96] (Honeywell Inc.) et Koala (Philips) [van Ommering& 00]. Chronologiquement, les premiers ADLs, dit de première génération, sont spécifiques à un domaine particulier, avant l'apparition des ADLs de deuxième génération plus en mesure de couvrir plusieurs domaines.

1.2. Concepts fondamentaux

Les concepts de base d'une description architecturale sur lesquels s'accordent l'ensemble des travaux portant sur les ADL reposent sur les concepts de : Composant, Connecteur, Configuration et Interface. Un ADL doit fournir ainsi les notations et les sémantiques nécessaires pour la spécification de ces concepts de base.

De manière informelle, ces concepts sont illustrés ci-après :

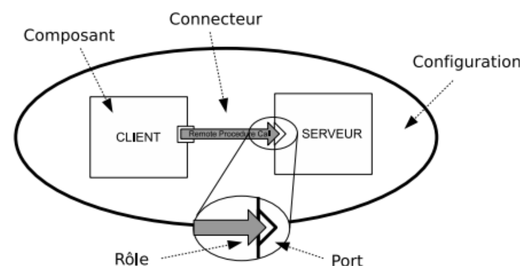


Figure 6 : Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL.

1. ***Composant :*** le composant représente le principal élément de calcul et de stockage des données dans un système. Un composant possède un ensemble d'interfaces, appelés ports, qui définissent les points d'interaction entre cet élément et son environnement.
2. ***Connecteur :*** le connecteur identifie l'interaction entre les composants. Le connecteur peut représenter une simple interaction comme une invocation à un service, ou bien un protocole complexe. Le connecteur peut disposer d'une glu qui définit comment les rôles interagissent entre eux, lorsqu'il est considéré comme une entité de première classe.
3. ***Interface :*** une interface constitue le « portail » d'interaction des composants, des connecteurs et des configurations. L'interface est aussi connue sous le nom de port pour les composants et les configurations, et de rôle pour les connecteurs.
4. ***Configuration :*** une configuration architecturale (ou simplement architecture ou système) est un graphe qui montre la façon dont un ensemble de composants sont reliés les uns aux autres par l'intermédiaire de connecteurs.
5. ***Niveaux d'abstraction d'une description d'architecture logicielle :*** la spécification d'une architecture logicielle par un ADL peut passer par plusieurs niveaux d'abstraction.

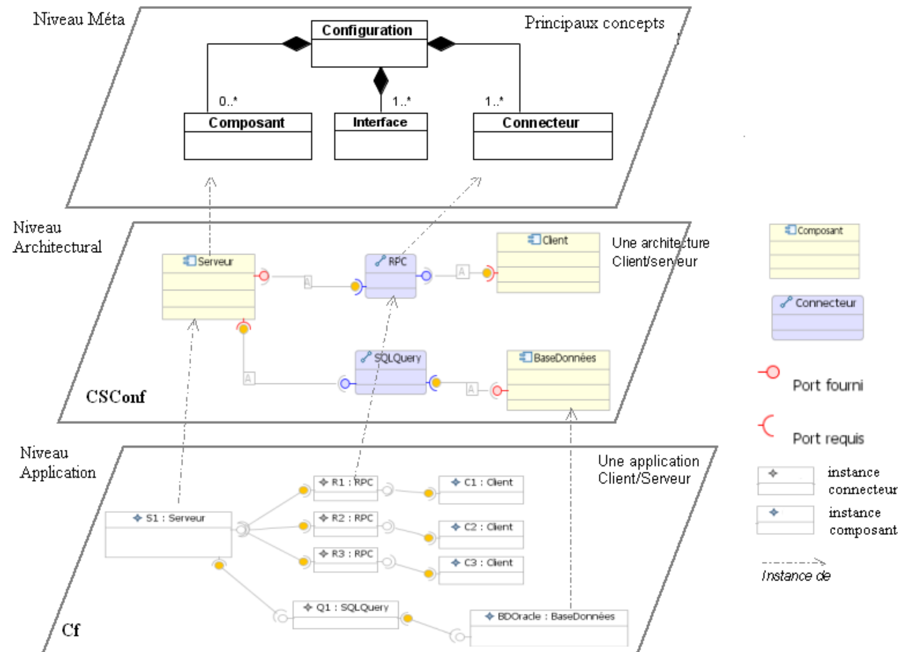


Figure 7 : Illustration des niveaux d'abstractions de description d'architectures logicielles.

Le contenu de chaque niveau dépend des concepts réifiés par cet ADL. En effet, nous constatons que dans tous les ADL le concept composant est réifié et considéré comme entité de première classe. La majorité des ADL distinguent le composant type du composant qui représente l'instance de ce composant type. La majorité des ADL distinguent deux niveaux de description pour les composants, le niveau type et le niveau instance. Cette distinction n'est pas considérée pour tous les autres concepts. De manière générale, dans les travaux menés, nous considérons que la description d'une architecture logicielle gagne à être établie au travers de trois niveaux d'abstraction, permettant d'introduire et de gérer les concepts. Ces niveaux sont : le niveau Méta, le niveau Architectural et le niveau Application illustrés par la Figure 7.

1.3. Survol des travaux d'évolution structurelle d'architectures logicielles à base de composants

Les travaux menés avec mes collaborateurs abordent l'évolution des systèmes logiciels à base de composants au niveau de la description de leur architecture. Ils s'adressent exclusivement aux architectes logiciels et aucunement aux utilisateurs finaux. Nos travaux offrent une assistance d'aide à l'évolution aux architectes. Ainsi, de la même façon que l'architecture logicielle permet de raisonner sur les propriétés fonctionnelles et non fonctionnelles d'un système à un haut niveau d'abstraction, elle permet à l'architecte de raisonner sur les différentes évolutions d'un système en termes d'ajout et de suppression de composants, de connecteurs ou sur leur réorganisation sans pour autant se noyer dans la conception dans un paradigme donné ni le code source du système.

Je considère en tant qu'artefact tout élément de l'architecture logicielle pouvant être amené à évoluer. Ainsi, il faut identifier ce qui peut évoluer, comment le faire évoluer, comment garantir la cohérence de l'évolution et de de l'architecture ayant évolué et ensuite comment répercuter ces évolutions de l'architecture logicielle sur le système qu'elle décrit. Je suis convaincue de l'importance d'intégrer l'analyse et la spécification de l'évolution pour toute architecture logicielle, et ce à n'importe quel

stade de son cycle de développement et de son cycle de vie. En effet, par ses spécificités, une architecture logicielle est appelée à être réutilisée, adaptée, maintenue et déployée. Son évolution fait partie intrinsèque de sa vie.

Travaux fondateurs : les travaux existants et dédiés à l'évolution des architectures logicielles à base de composants sont généralement attachés aux langages de descriptions d'architectures, les ADLs (*Architecture Description Languages*). Aussi, la gestion de l'évolution reste spécifique à chaque ADL. Il existe deux catégories d'ADL : les ADL spécifiques à un domaine, qualifiés d'ADL de *première génération* tels que Darwin [Magee& 95], Wright [Allen& 96], C2SADEL [Medvidovic& 99], C2 [Medvidovic& 96] et les ADL qui peuvent adresser plusieurs domaines, donc plus génériques, qualifiés d'ADL de *deuxième génération* tels que ACME [Garlan& 00], xADL2.0 [Dashofy& 01], SafArchie [Barais 05], UML2.0 [OMG 03], Mae [Hoeck 05]. Ces différents ADLs offrent tous la possibilité d'ajouter et de supprimer les instances des concepts de base. Ces supports de l'évolution sont donc *implicites* ou *prédéfinis*, selon les ADLs. Outre ces supports de l'évolution, Wright et C2SADEL permettent une évolution *explicite* sur une partie des concepts par la possibilité de définir des contraintes. Quant à Darwin, seuls les composants et les configurations peuvent évoluer de manière *prédéfinie*. Concernant les ADLs de seconde génération, du fait qu'ils soient plus génériques, proposent des mécanismes explicites pour l'évolution. Les plus notables sont SafArchie et xADL 2.0.

Analyse : les ADLs cités permettent, sur certains concepts, variant d'un ADL à un autre, d'effectuer des évolutions implicites ou préfinies, essentiellement dans les ADLs de première génération. Les ADLs de seconde génération sont plus souples du fait qu'ils proposent plus de supports explicites pour l'évolution, essentiellement au travers de l'expression de contraintes, d'invariants et de contrats. Cependant, ces supports, qu'ils soient prédéfinis ou explicites, sont applicables à certains concepts-types et/ou aux instances de certains de ces concepts-types. Il n'y a donc pas de généralisation ni d'approches génériques.

Constat : la principale défaillance qui ressort est l'absence d'uniformité dans la spécification des concepts et des supports de l'évolution : tous les concepts ne sont pas considérés comme des entités de première classe, et ne peuvent par conséquent pas être manipulés et encore moins évolués. Seuls les concepts réifiés en tant qu'entités de premières classes peuvent être amenés à évoluer. De plus, certains supports de l'évolution sont définis au sein du système. Cependant, du fait du caractère de plus en plus évolutif des systèmes, les actions d'évolution sont de plus en plus récurrentes. Malheureusement, il n'existe pas de moyens d'identifier et de réutiliser un savoir-faire d'évolution d'un système à un autre. Je considère qu'il s'agit des principaux verrous scientifiques à traiter : spécifier l'évolution en la dissociant du comportement des concepts et proposer des concepts et processus propres à l'évolution au sein de méta-modèles permettant sa spécification, son utilisation et sa réutilisation.

Les quatre prochaines sections de ce chapitre présentent nos travaux et contributions :

2. Un modèle pour l'Evolution Structurelle dans les Architectures Logicielles

2.1. Fiche d'identité

Titre :	SAEV : un modèle pour l'Evolution Structurelle dans les Architectures Logicielles
Paradigme(s) :	Architectures logicielles à base de composants – ADLs
Principales contributions :	. Proposition d'un modèle d'évolution dénommé SAEV (Software Architecture EVolution model) permettant l'abstraction, la spécification et la gestion de l'évolution des architectures logicielles. SAEV se veut un modèle générique, uniforme et indépendant de tout langage de description d'architectures logicielles. . Enrichissement de la sémantique des connecteurs avec des propriétés sémantiques pour propager automatiquement les impacts d'une évolution.
Mots-clés :	Architecture logicielle à base de composants, stratégie d'évolution, règle d'évolution, impacts et propagation des impacts, propriétés sémantiques des connecteurs.
Collaborateur(s) :	N. Sadou (doctorante), M. Oussalah (directeur de thèse, Université de Nantes), B. Turcaud (Master 2, Université de Nantes), X. Seignard (Master2 / Ingénieur, Polytech'Nantes)
Mon implication :	Co-encadrement à 50 % de la Thèse de N. Sadou, à 50% du Master 2 de B. Turcuad et de 100% du Master 2 de X. Seignard.
Cadre de collaboration :	Fonds personnels de la doctorante.
Période :	2003-2007
Principales publications :	1 revue nationale ; 2 conférences de rang A : COMPSAC 2007, ER2005 ; 2 conférences de rang B : SEKE 2006, SERP 2005 ; 4 conférences/ateliers internationaux ; 2 conférences nationales.

Figure 8 : Fiche synthétique de SAEV.

2.2. Synthèse

La problématique abordée est l'évolution structurelle dans les architectures logicielles à base de composants. Un modèle d'évolution d'architectures logicielles baptisé SAEV (*Software Architecture EVolution model*) est proposé en réponse à la problématique de manque de solutions uniformes et décorrélées des ADLs. SAEV a été réfléchi et conçu comme un modèle d'évolution indépendant de tout ADL, pouvant s'appliquer à toute architecture logicielle, quel que soit son langage de description à condition de disposer du méta-modèle de l'ADL.

1. **L'évolution, une préoccupation à part entière :** la principale contribution de SAEV est de permettre d'accroître l'évolutivité structurelles des architectures logicielles à base de composants. Pour ce faire, l'approche adoptée est la séparation totale des deux préoccupations que sont l'architecture logicielle et son évolution. L'évolution d'une architecture logicielle est considérée comme un système à part entière. SAEV offre une abstraction de l'évolution en la spécifiant explicitement et indépendamment de la spécification de l'architecture elle-même et surtout de son comportement. SAEV permet donc l'abstraction, la spécification et la gestion de l'évolution des architectures logicielles. Il est un modèle générique, uniforme et indépendant de tout langage de description d'architectures logicielles.

2. **Rôle accru des connecteurs dans la propagation de l'évolution :** la seconde contribution de SAEV est de faire jouer un rôle de premier plan aux connecteurs dans la propagation et le contrôle des impacts d'évolution d'un élément architectural aux autres éléments auxquels ils sont reliés, tout en sauvegardant la cohérence globale de l'architecture. Cette propagation a un équilibre subtil à trouver entre une partie automatisée et contrôlée qui évite d'introduire des incohérences dans l'architecture et une partie d'interactions appropriées avec l'architecte. SAEV propose ainsi d'enrichir la description d'une architecture logicielle avec des propriétés sémantiques sur le degré de corrélation entre ses éléments. Ces propriétés sont associées au concept de *connecteur*.

2.3. Verrous scientifiques

Dans ces travaux, en nous basant sur les définitions fondatrices de la discipline (confère Chapitre Chapitre 2. 2.). Les deux verrous scientifiques abordés sont :

1. **Support à l'évolution :** le verrou scientifique abordé est l'absence de supports d'évolution explicites, génériques et utilisables par toute architecture logicielle quel que soit l'ADL support. Cette absence impose de développer des supports d'évolution au cas par cas. La réutilisation s'en trouve restreinte dans un seul et même contexte et ADL. La capitalisation et la mutualisation s'en trouvent également limitées.
2. **Propagation de l'évolution dans une architecture :** le verrou scientifique abordé est la faible expressivité et le manque de sémantique des différents degrés de corrélation que portent les connecteurs entre les éléments architecturaux qu'ils relient.

Les contributions sont présentées ci-après :

2.4. SAEV : un modèle d'évolution d'architectures logicielles

a. *Objectif*

L'objectif principal de SAEV (Software Architecture EVolution model) a été de proposer un modèle d'évolution d'architectures logicielles applicable de manière uniforme aux trois niveaux d'abstraction considérés (Figure 7). Le principe est de rendre une architecture logicielle réactive de manière automatique aux évènements étiquetés comme étant des événement déclencheurs d'évolution. Grâce à cette approche événementielle, toute évolution peut automatiquement se déclencher et se propager. SAEV vise ainsi à répondre aux objectifs suivants :

- Offrir un niveau d'abstraction de l'évolution des architectures logicielles, en dissociant complètement le traitement de l'évolution des éléments d'une architecture logicielle, de leurs comportements proprement dit. Il s'agit en effet de spécifier explicitement l'évolution pour pouvoir la traiter explicitement et indépendamment de la spécification des éléments architecturaux eux-mêmes, et donc indépendamment de leurs langages de description.
- Permettre la spécification et la gestion d'une manière uniforme de l'évolution de tout élément architectural pouvant être réifié par un ADL (composant, connecteur, configuration, etc.) et à travers les différents niveaux d'abstraction de description d'architectures logicielles (Figure 7).
- Prendre en compte explicitement la propagation des impacts de l'évolution d'un élément architectural sur les autres éléments concernés de l'architecture, tout en sauvegardant la cohérence globale de cette architecture.

- Favoriser la réutilisation des spécifications d'évolution. En effet, le modèle SAEV doit faciliter au concepteur la spécification des évolutions qu'il souhaite répercuter sur l'architecture en lui permettant de réutiliser des spécifications d'évolution déjà définies.

b. *Méta-modèle et concepts de SAEV*

Le méta-modèle de SAEV est présenté dans le diagramme de classe de UML (Figure 9) :

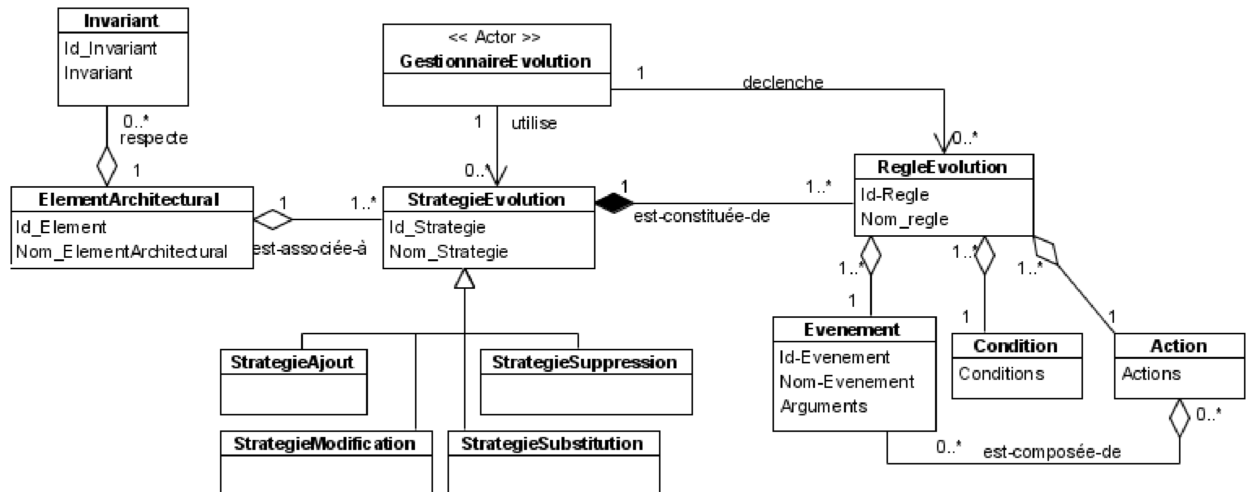


Figure 9 : Méta-modèle de SAEV.

Les principaux concepts de SAEV sont :

1. *Élément architectural :* modélise et réifier tout élément d'une architecture susceptible d'évoluer : une configuration, un composant, un connecteur, une interface ou tout autre concept supporté par l'ADL qui a servi à la description de l'architecture logicielle. Ces éléments concernent les trois niveaux d'abstraction Méta, Architectural et Application.
2. *Invariant :* un invariant représente une contrainte structurelle sur un élément architectural ou sur l'ensemble de l'architecture. Tout changement dans l'architecture logicielle doit garantir le maintien de chaque invariant pour sauvegarder la cohérence globale de l'architecture.
3. *Règle d'évolution :* les règles d'évolution sont des règles ECA (Evènement/Condition/Action) [Dayal& 88]. Chaque règle encapsule une *action* d'évolution (ajout/ suppression/ modification/ substitution), son *évènement* déclencheur et les *conditions* nécessaires à son déclenchement. Une règle est élue dynamiquement dès la réception de l'évènement qui lui est associé. Une action est une opération élémentaire à exécuter ou un évènement (une référence vers une autre règle). Le formalisme ECA permet ainsi de spécifier indifféremment le déclenchement automatique d'une règle d'évolution que de la propagation des impacts d'une évolution.
4. *Stratégie d'évolution :* une stratégie d'évolution regroupe l'ensemble des règles permettant de spécifier l'évolution d'un élément architectural par catégories d'évolution : ajout/ suppression/ modification/ substitution. D'autres stratégies peuvent être définies.
5. *Gestionnaire d'évolution :* il est le chef d'orchestre. Son rôle est d'intercepter tout évènement d'évolution émanant de l'architecte ou des règles d'évolution vers l'architecture ou l'un de ses éléments, puis d'identifier la stratégie d'évolution correspondante. Suivant cette stratégie, il déclenche la ou les règles d'évolution associées à l'évènement reçu. Il est chargé ensuite de

propager les impacts des règles d'évolution exécutées. Le gestionnaire d'évolution doit vérifier aussi le maintien des invariants et la cohérence globale de l'architecture logicielle.

c. *Illustration des règles*

La convention utilisée pour spécifier des règles d'évolution est donnée par la table suivante :

Tableau 1 : Notation et illustration d'une règle.

Elément d'une règle	**Notation**	**Exemple**
Désignation	**Ri: Règle de {Opération d'Evolution}{Elément architectural}** i : représente une chaîne de caractère	R1 : Règle de suppression Composant
Evénement	**Opération Évolution-Élément Architectural (Elt : Elément Architectural, [autres paramètres]**)	**Supprimer-Composant(C : Composant, Cf : Configuration)**
Condition	Sous forme d'une conjonction de prédicats sur les éléments architecturaux ou sur des ensembles de ces éléments architecturaux	**Composants(Cf)** : l'ensemble des composants de la configuration **Cf.** **C € composants(Cf)** : prédicat à **vrai** si le composant **C** est un composant de la configuration **Cf**.
Action	Soit un événement, donc noté comme tel, soit une *opération élémentaire*	***Elt.Execute. Operation Evolution([Paramètres])***

Exemple complet : le Tableau 2 illustre le concept de règle :

Tableau 2 : Une règle d'évolution SAEV.

R1 :Règle-Suppression-Composant
Evenement : supprimer-composant(C : composant, Cf : Configuration) ;
Condition : C $\in$ Composants(Cf), $\exists$ B$\subset$ Bindings(Cf,C) et B$\neq \phi$;
Action : **Pour tout** b $\in$ B Supprimer-binding(b,Cf,C) ; (1) **FinPour** **Pour tout** I $\in$ interface-composant(C) Supprimer-interface-composant(I,Cf,C) ; (2) **FinPour** Modifier-configuration(Cf, C) ; (3) *C.Execute-supprimer-composant(Cf).* (4)

La règle *R1* décrit la suppression du composant *C* qui est à l'extrémité d'une configuration *Cf*, exprimée par la condition suivante qui indique, qu'il existe des bindings entre la configuration *Cf* et le composant *C* :

$$(\exists B \subset Bindings(Cf, C) et B \neq \phi)$$

La partie action de cette règle indique que pour la suppression de ce composant, il faut en premier supprimer (1) tous ses liens (bindings), puis (2) ses interfaces, (3) supprimer le composant de la liste des éléments de la configuration avant de définitivement le (4) supprimer. Les trois premières actions de *R1* correspondent réellement à des évènements qui déclenchent des règles d'évolution relatives respectivement à un *binding*, une *interface composant* et une *configuration*. La dernière action invoque le code source qui implémente l'opération de suppression d'un composant.

Ces règles sont les actions clés du processus d'évolution explicités ci-après :

d. Processus d'évolution

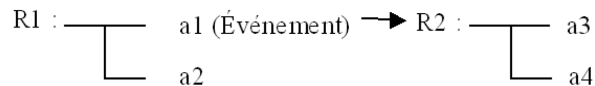
1. *Phase de spécification de l'évolution :* spécifier les éléments de l'architecture à faire évoluer, selon le niveau d'abstraction considéré. Pour chaque élément architectural, il faut spécifier : ses invariants, ses stratégies d'évolution, ses règles d'évolution associées à chacune des stratégies d'évolution.
2. *Phase d'exécution d'une évolution :* le mécanisme opératoire est illustré dans la Figure 10, via le diagramme d'activité UML. Ce mécanisme opératoire est composé de quatre principales étapes :

   Etape1 : *Interception d'un évènement d'évolution*. Il s'agit d'un événement déclenché par l'architecte ou un événement lié à une propagation d'évolution.

   Etape 2 : *Recherche de la stratégie d'évolution*. Il s'agit de détecter le sous-ensemble de règles d'évolution éligibles.

   Etape 3 : *Recherche de la règle d'évolution à exécuter* au sein de la stratégie détectée.

   Etape 4 : *Exécution des règles d'évolution et la propagation des impacts :* chaque action d'une règle est exécutée ainsi que ses propagations, avant de passer à l'action suivante. Par exemple, si une règle R1, constituée des actions a1, a2 est exécutée. Si a1 correspond à un évènement qui déclenche la règle R2, alors toutes les actions de R2 (a3, a4) doivent être exécutées avant de passer à l'action a2 de R1.



   Une règle se trouve dormante jusqu'à l'occurrence de l'évènement qui la compose. Quand l'évènement a eu lieu, et si les conditions sont satisfaites, alors la partie action est exécutée apportant une réponse appropriée à l'évènement.

   Etape 5 : *Vérification de la cohérence et validation de l'évolution* : la vérification de la cohérence se fait au vu des invariants. Deux modes sont proposés : le *mode immédiat* qui vérifie la cohérence après chaque exécution d'une opération d'évolution élémentaire d'une règle et le *mode différé* qui ne vérifie la cohérence qu'une fois toutes les actions de la règle exécutée ainsi que leurs propagations éventuelles.

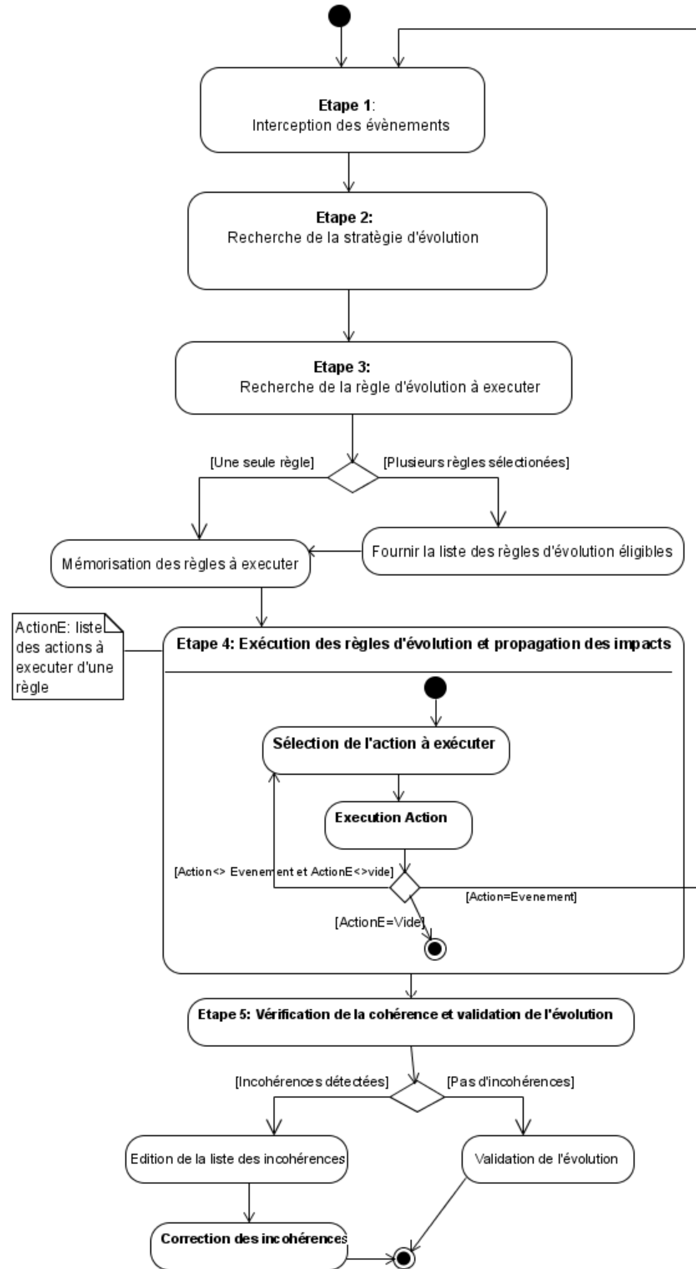


Figure 10 : Processus opératoire de SAEV.

e. Conclusion

Le modèle SAEV a été proposé pour dissocier la spécification de l'évolution de la spécification de l'architecture logicielle elle-même. C'est ainsi que SAEV propose notamment les concepts de *règle d'évolution*, de *stratégie d'évolution* et d'*invariants*. Ces concepts permettent la spécification et la gestion de l'évolution d'une architecture logicielle, indépendamment de tout ADL. SAEV considère les différents niveaux d'abstraction de description d'une architecture logicielle, et la nécessité de gérer l'évolution au travers de ces différents niveaux d'abstraction de manière uniforme. En effet, dans SAEV, les évolutions d'une architecture logicielle sont spécifiées sur la base des éléments d'un niveau d'abstraction et elles s'appliquent à l'évolution des éléments du niveau inférieur. Ainsi une

évolution est spécifiée au niveau Méta pour gérer l'évolution au niveau Architectural, et une évolution est spécifiée au niveau Architectural pour gérer l'évolution au niveau Application. Il est alors nécessaire de bien identifier le niveau d'abstraction de l'architecture à faire évoluer.

2.5. Propriétés sémantiques des connecteurs

Dans les cas où plusieurs règles d'évolution sont éligibles pour la propagation des impacts, le modèle SAEV peut ne pas déterminer automatiquement la règle d'évolution à déclencher. Le recours à l'architecte est alors indispensable pour effecteur ce choix. Le but est de répondre à cette limite en permettant à SAEV d'automatiser autant que possible la détermination et la propagation des impacts d'une évolution.

a. Objectif

Le connecteur peut jouer un rôle prépondérant dans la propagation des impacts d'une évolution au sein d'une architecture logicielle. Notre objectif est d'enrichir le concept de connecteur avec des propriétés sémantiques. Évidemment, seuls les ADL qui réifient et qui considèrent le concept de connecteur comme entité de première classe peuvent exploiter ces propriétés sémantiques.

Un connecteur est un bloc de constructions utilisé pour modéliser les interactions entre les composants. Un connecteur relie précisément l'interface fournie d'un composant à l'interface requise d'un autre composant (confère Chapitre 2).

b. Propriétés sémantiques

Nous définissons une propriété sémantique comme la propriété associée à un connecteur pour exprimer le degré de corrélation ou de dépendance qui existe entre les deux interfaces composant fournie et requise qu'il relie. Nous notons la propriété *P* définie par un connecteur *N* entre l'interface source *Is* et l'interface cible *Ic* par *PN : Is, Ic* où (confère Figure 11) :

- Is : interface composant fournie attachée à un connecteur sous le terme d'interface source du connecteur ;
- Ic : interface composant requise attachée à ce même connecteur sous le terme d'interface cible du connecteur.

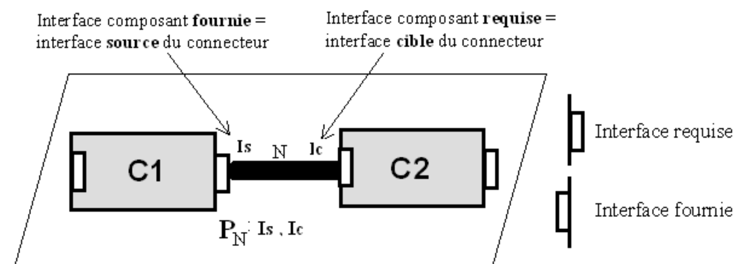


Figure 11: Illustration d'une propriété sémantique.

Les propriétés sémantiques des connecteurs sont spécifiées au niveau *Méta*. Les propriétés seront ainsi renseignées à la spécification de l'architecture au niveau *Architectural* et s'appliqueront à l'évolution d'une architecture au niveau *Application*.

En s'inspirant notamment des travaux sur la sémantique des liens de composition et d'héritage entre les classes dans le paradigme objet [Kim& 89] utilisées pour la propagation des changements dans les schémas de classes, nous proposons les propriétés sémantiques suivantes :

- **Exclusivité / Partage :** la propriété d'exclusivité/partage exprime les contraintes sur les échanges de services entre les interfaces. Ainsi, un connecteur définissant la propriété d'*exclusivité (E)*, dit *connecteur exclusif*, indique que les deux interfaces source et cible de ce connecteur interagissent uniquement entre elles. A l'inverse, un connecteur définissant la propriété de *partage (P)*, dit *connecteur partagé* indique que les interfaces source et cible de ce connecteur peuvent interagir avec un nombre quelconque d'interfaces.
- **Dépendance / Indépendance :** cette propriété exprime une dépendance existentielle ou non entre les interfaces composants fournies ou requises reliées par le connecteur. La propriété *Dépendance* se décline en : *Dépendance forte (DF) :* l'existence de l'interface cible du connecteur est complètement liée à l'existence de son interface source ; *Dépendance faible(Df)* : plusieurs autres connecteurs de dépendance faible peuvent être reliés à l'interface cible de ce connecteur. L'*Indépendance (I)*, à l'inverse, indique que l'existence de l'interface cible du connecteur est indépendante de l'existence de son interface source.
- **Prédominance / Non prédominance :** la propriété de prédominance indique s'il existe ou non une dépendance existentielle de l'interface source d'un connecteur vis à vis de son interface cible. Cette propriété est symétrique à la propriété de dépendance. En effet, la prédominance met l'accent sur le fait que l'existence d'une interface source d'un connecteur est liée à l'existence de son interface cible, alors que la dépendance exprime le fait que l'existence de l'interface cible dépend de l'existence de l'interface source de ce connecteur. La propriété de *Prédominance* se décline en *Prédominance forte(PRF)*qui indique que l'existence de l'interface source du connecteur est fortement liée à celle de l'interface cible de ce connecteur et en *Prédominance faible(PRf)* indique que plusieurs autres connecteurs peuvent être attachés à l'interface source de ce connecteur.
- **Cardinalité /Cardinalité inverse :** la *Cardinalité* d'un connecteur indique le nombre d'instances de l'interface cible de ce connecteur qui peuvent être attachées à une même instance de l'interface source de ce connecteur. La *Cardinalité inverse* d'un connecteur indique le nombre d'instances de l'interface source qui peuvent être attachées à une même instance de l'interface cible de ce connecteur. La cardinalité et la cardinalité inverse peuvent être exprimées sous forme d'une valeur fixe ou sous forme d'un intervalle en indiquant la valeur minimale et la valeur maximale de cet intervalle.

Pour chacune de ces propriétés, nous avons définis des contraintes de définition de la propriété, une spécification formelle, un exemple illustratif, les valeurs par défaut de chaque propriété, des restrictions associées aux valeurs des propriétés sémantiques. Les propriétés sémantiques peuvent être exprimées de manière non-ambiguë sur les connecteurs d'une architecture logicielle.

c. Conclusion

Les connecteurs ont été enrichis de propriétés sémantiques afin de leur donner une plus grande capacité d'expressivité de contraintes pour augmenter la capacité d'automatisation de la propagation des évolutions. Ces propriétés sont Exclusivité / Partage, Dépendance / Indépendance et Prédominance / Non prédominance. Elles se basent sur les travaux sur la sémantique des liens de composition et d'héritage entre les classes dans le paradigme objet [Kim& 89]

2.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Encadrement doctoral (confère *section 7. Détails de l'encadrement doctoral* du CV détaillé) :

1. **2003-2007 :** thèse de Nassima SADOU (Doctorat en informatique de l'Université de Nantes), SAEV, un modèle d'évolution dans les architectures logicielles à base de composants » *- Taux d'encadrement : 50%.*
2. **2007** : Xavier SEIGNARD, Master 2 de l'Université de Nantes et ingénieur de l'Ecole Polytechnique de Nantes, « Prototypage de SAEV : approche par transformation de graphes » *- Taux d'encadrement : 100%.*
3. **2005** : Brice TURCAUD, Master 2 de l'Université de Nantes, « Validation de SAEV : un modèle d'évolution d'architectures logicielles à base de composants », *- Taux d'encadrement : 50%.*

2.7. Brève synthèse des travaux fondateurs

La plupart des ADL considèrent l'évolution des composants types et des configurations. Pour les connecteurs et interfaces types, seul les ADL qui les réifient et les considèrent comme entités de premières classes abordent leurs évolutions tels que Mae et xADL2.0. Peu d'ADL aborde l'évolution des composants, des connecteurs et des interfaces. La plupart des ADL considèrent uniquement l'évolution statique des architectures logicielles ; Les ADL tels que Dynamic ACME, Dynamic Wright abordent l'évolution dynamique anticipée d'une architecture logicielle. Autrement dit, ils permettent d'exprimer des besoins d'évolutions prévus et préalablement connus. Aucun des ADL étudiés ne permet de refléter automatiquement les évolutions de l'architecture vers le code source. De ce que nous avons étudié, seuls les outils ArchStudio (associé à C2SADEL) et ArchEvol ont tenté de répondre à cette préoccupation. Pour l'évolution statique des composants et des connecteurs types, la plupart des ADL se limitent au mécanisme de composition hiérarchique. D'autres ADL tels que C2SADEL et Mae proposent aussi le sous-typage hétérogène. Des ADL étudiés seul SafArchie propose des opérations d'évolutions ajout/suppression/ déplacement pour faire évoluer les composants et les interfaces. xADL2.0 et Mae adoptent le versionnent pour faire évoluer une architecture logicielle.

2.8. Positionnement et bilan

La principale contribution des travaux autour de SAEV a été de découpler l'évolution du comportement d'une architecture et de son ADL et de la spécifier en conséquence au sein d'un méta-modèle : SAEV, pour *Software Architecture EVolution model*. Il permet l'abstraction, la spécification et la gestion de l'évolution des architectures logicielles. Ce modèle a pour principales caractéristiques d'être uniforme, générique et extensible. Il considère que tout concept est appelé à évoluer. Le cœur de SAEV est le concept de *règle d'évolution*, qui est de type ECA (Evènement – Condition – Action). Toute évolution d'un concept donné est encapsulée dans des règles, qui sont rattachées à des *stratégies d'évolution* (ajout, suppression, modification…) et qui peuvent être étendues. Ces stratégies sont associées à un seul concept à la fois, au même titre que ses invariants. Le processus d'évolution de SAEV a une approche événementielle : dès qu'un événement d'évolution survient sur un concept, la règle d'évolution adaptée est automatiquement sélectionnée et exécutée. Les impacts de son exécution sont gérés sur le même principe événementiel : par le déclenchement des règles appropriées.

La seconde contribution de ce travail a été d'enrichir le concept de *connecteur* avec des propriétés sémantiques, qui ont pour principal objectif de renseigner le degré de couplage entre deux composants. Grâce aux informations portées par ces propriétés, le degré d'automatisation de la propagation des impacts d'une évolution a été augmenté et mieux contrôlé.

Validation : SAEV a eu des validations distinctes :

1. La validation théorique de SAEV s'est faite en deux temps :
    a. *Appliqué à un ADL :* l'ADL utilisé a été l'ADL COSA (Component-Object based Software Architecture) [Smeda 06] développé dans l'équipe et ayant l'avantage d'avoir les différents niveaux d'abstraction dont le niveau Méta indispensable pour spécifier les concepts.
    b. *Appliqué dans le cadre du projet Etat-Région Metedi :* dédié à l'apprentissage à distance et les différents scénarios pédagogiques. Ils ont été modélisés sous forme d'architectures logicielles (confère Section 6.3.2 du CV détaillé). SAEV a été utilisé comme moyen de faire évoluer ces scénarios, notamment en explicitant les différentes évolutions possibles d'un scénario au sein de règles d'évolution.
2. La validation pratique : a été l'objet de deux sujets de Master 2 :
    a. De B. Turcaud (2005) : en implémentant le méta-modèle avec une approche orientée objet.
    b. De X. Seignard (2007) : le gestionnaire d'évolution a été prototypé en utilisant approche par transformation de graphes.

Ma contribution : mon implication dans SAEV a été le co-encadrement de la thèse de Nassima Sadou à hauteur de 50% avec Mourad Oussalah et des publications associées, du co-encadrement à 50% d'un premier master-recherche (de Brice Turcaud) sur le premier prototype des concepts de SAEV et de l'encadrement à 100% d'un second master-recherche (de Xavier Seignard) pour explorer l'utilisation de la transformation de graphes pour la propagation d'impacts au travers de l'outil AGG[10]. J'ai également participé à la compréhension et à la spécification des AL et de leur évolution dans le cadre de Metedi.

Valorisation : ces travaux ont donné lieu à 12 publications, dont deux conférences de rang A ER 2005 et COMPSAC 2007 (références [32] et [33] du CV détaillé) et deux conférences de rang B, SEKE 2006 et SERP 2005 (références [46] et [51] du CV détaillé).

---

10 http://tfs.cs.tu-berlin.de/agg/

3. Capitalisation et réutilisation d'évolutions : styles d'évolution

3.1. Fiche d'identité

Titre :	SEAM : Spécification et capitalisation d'expertise de mise à jour dans les architectures logicielles à base de composants : une approche par les styles d'évolution
Paradigme(s) :	Architectures logicielles à base de composants – ADLs
Principales contributions :	. Proposition du modèle d'évolution SAEM (Style-based Architectural Evolution Model) pour capitaliser les évolutions récurrentes à des fins de réutilisation. Il propose le concept de *style d'évolution*. SAEM est un modèle d'évolution générique, uniforme et indépendant de tout langage de description d'architecture. . Approche de réutilisation au travers d'une démarche pour la construction de bibliothèques pour les styles d'évolutions. Les bibliothèques sont élaborées selon deux types de processus complémentaires : « pour la réutilisation » et « par la réutilisation ». Nous utilisons une technique de raisonnement classificatoire.
Mots-clés :	Style d'évolution, Patron d'évolution, Réutilisation de l'évolution, Bibliothèque, Architecture Logicielle, Langage de Description d'Architecture.
Collaborateur(s) :	O. Le Goaer (doctorant), M. Oussalah (directeur de thèse, Université de Nantes), D. Seriai (co-encadrant, Ecole des Mines de Douai).
Mon implication :	Co-encadrement à 40% de la Thèse de O. Le Goaer.
Cadre de collaboration :	Bourse Ministère.
Période :	2005-2009.
Principales publications :	1 chapitre d'ouvrage ; 1 conférence de rang A : IEEE COMPSAC 2008; , 7 conférences de rang B, SEKE 2008-2010, SEDE 2010, SERP 2010, IEEE IRI 2006, ICSOFT 2007 ; 3 autres conférences/ateliers internationaux ; 1 atelier national : Rimel 2007.

Figure 12 : Fiche synthétique de SAEM.

3.2. Synthèse

Nous proposons le concept de *Style d'évolution* comme un moyen pour capitaliser le savoir-faire d'évolution à des fins de réutilisation. Il s'agit d'un patron pour mettre à jour des architectures logicielles à base de composants. L'objectif est de capturer des modèles récurrents et réutilisables d'évolutions architecturales.

Il cible la capitalisation et la réutilisation d'activités d'évolution d'architectures logicielles décrites à l'aide des ADLs. Un style d'évolution est une solution réutilisable à un problème d'évolution bien identifié, et ré-applicable à différentes situations. L'idée principale est de capitaliser tout savoir-faire d'évolution éprouvé afin de pouvoir :

- Le réutiliser dans toute situation d'évolution à laquelle il est adapté.
- L'utiliser pour aider à définir un nouveau savoir-faire d'évolution ayant des similarités avec de nouveaux besoins d'évolution.

Le concept de style d'évolution s'inspire pleinement de la notion de style architectural, notion fondamentale pour les ADLs. Un style architectural peut être vu comme un patron au niveau architectural (tels que les styles client/serveur, pipe&filter, style en couches…). La notion de style architectural a été introduite pour capitaliser des savoir et des savoir-faire pour la conception d'architectures. Perry et Wolf définissent un style architectural comme « ce qui abstrait les éléments et les aspects formels de diverses architectures spécifiques » [Perry& 92]. Shaw et Garlan utilise le terme de style architectural pour désigner une famille de systèmes (c'est-à-dire d'architectures applicatives) qui partagent un vocabulaire commun de composants et connecteurs, et qui répondent à un ensemble de contraintes pour ce style [Shaw& 96]. Le concept de style d'évolution proposé s'inspire également des Design Patterns [Gamma& 94] définis pour aider à capitaliser des savoir-faire de conception et des Styles architecturaux [Garlan 95] destinés à recenser, formaliser et réutiliser des organisations logicielles profitables.

3.3. Verrous scientifiques

Le principal verrou abordé est le peu de travaux dédiés à la réutilisation de l'évolution architecturale. Les modèles d'évolution existants ne proposent pas de spécifier et de gérer l'évolution de manière explicite. Notamment, la notion de réutilisation de l'évolution est très peu abordée. Pourtant, la capitalisation et la réutilisation via des solutions éprouvées est largement répandue et acceptée sur la partie conception. L'exemple le plus connu est celui des patrons de conception de Gamma [Gamma& 94] et des styles architecturaux [Garlan 95] [Shaw& 96].

Les deux verrous abordés sont :

- L'absence de modèle permettant l'abstraction, la spécification et la gestion du savoir-faire d'évolutions architecturales, qui soit générique, uniforme et indépendant de tout langage de description d'architecture.
- Le développement d'une approche de classification, de récupération et de réutilisation du savoir-faire d'évolution.

Les contributions sont présentées ci-après :

3.4. SAEM : Style-based Architectural Evolution Model

SAEM pose la définition suivant de *style d'évolution* : « *Un style d'évolution capture une manière caractéristique de procéder à l'évolution de tout ou partie d'une architecture logicielle. Il sert de guide pour un architecte qui doit alors se conformer au style*. » [Le Goaer 09].

a. Méta-modèle SAEM et principaux concepts

Une évolution est réifiée comme un style et attachée à chaque élément architectural, en tant que partie intégrante de sa description tout en étant distincte. Le méta-modèle de SAEM est formalisé au sein du diagramme de classes UML 2.0 suivant :

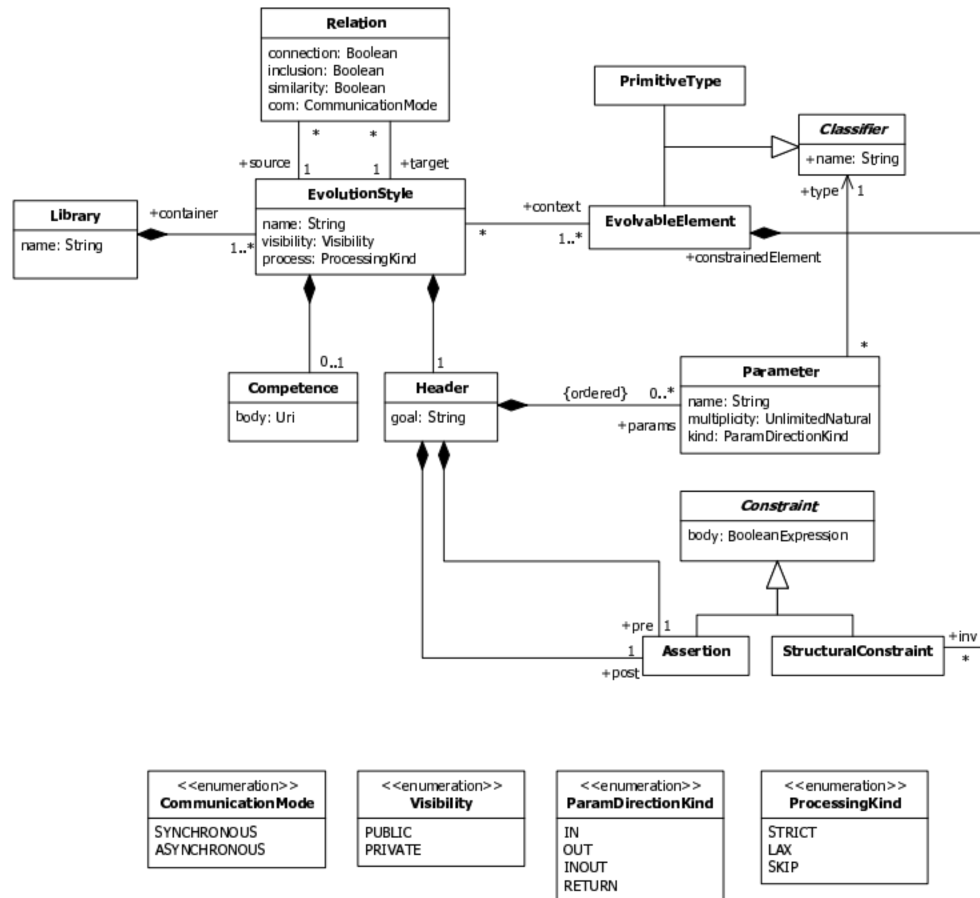


Figure 13 : Méta-modèle de SAEM.

Les principaux concepts de SAEM sont :

- *Élément évolutif* (*EvolvableElement*) : représente tout élément architectural appelé à évoluer (composant, connecteur, interface…). Il est nécessaire que cet élément architectural soit réifié afin qu'il puisse être traité.
- *Style d'évolution (EvolutionStyle) :* encapsule une évolution applicable à un élément évolutif. Il est constitué de trois parties :
  - *Nom :* qui identifie le style de manière unique.
  - *Entête* : obligatoire, elle possède une description informelle du but poursuivi et publie une liste de paramètres et d'assertions. L'élément évolutif apparaît comme un paramètre implicite nommé *context*.
  - *Compétence* : facultative. Quand elle n'est pas spécifiée, le style d'évolution est *abstrait*. Quand elle est spécifiée, elle décrit une implémentation de l'entête. L'implémentation (données, structures de contrôle) est une ressource externe, identifiable par son URI (Uniform Resource Identifier).

- *Exemple* : Le style *RemoveTail@Component* décrit la suppression d'un composant qui se trouve à l'extrémité de sa configuration d'appartenance (*self.context.owner*).

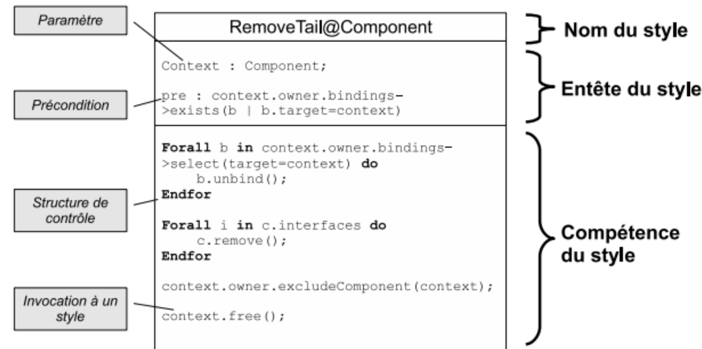


Figure 14 : Nom, entête et compétence d'un style d'évolution.

- *Contrainte (Constraint) :* permet de contrôler toute évolution. Il y a deux catégories de contraintes :
  - *Contrainte Structurelle (StructuralConstraint) :* représente une contrainte sur la structure d'un élément évolutif. Elle doit être respectée tout au long du cycle de vie de l'élément.

    Exemple : un composant de type *Client* ne possède pas plus de quatre ports de type *sendRequest*

    ```
    context Client
      inv : self.ports->select(isTypeOf(sendRequest))->size()<=4
    ```
  - *Assertion (Assertion)* : représente des contraintes sur l'état de tout ou partie de l'architecture avant, pendant, et après son évolution.

    Exemple : une assertion associée au style d'évolution *AddSendRequest*@Client. La précondition vérifie que le composant de type client possède un taux de requête strictement supérieur à la valeur 10. La postcondition vérifie que le composant possède bien un port de plus qu'avant son évolution.

```
context AddSendRequest@Client::invoke():void
  pre : self.context.requestRate > 10
  post : self.context@post.ports->size() = self.context@pre.ports->size()+1
```

- *Relation :* spécifie le lien entre styles d'évolution. Elle permet d'organiser les styles d'évolution entre eux pour faciliter leur découverte et leur classification. Elle est définie par un triplet d'éléments relationnels (a) de *connexion*, (b) d'*inclusion* et (c) de *similarité*, inspirés des travaux de Chaffin [Chaffin 89] et un mode de communication qui peut être *synchrone* ou *asynchrone.*
  Exemple : trois instances de *Relation* utilisées par SAEM, l'*utilisation* (ou *référence*), la *composition* et la *spécialisation*. :

Spécialisation : Relation	Composition : Relation	Utilisation : Relation
connection=true inclusion=true similarity=true com=#SYNCHRONOUS	connection=true inclusion=true similarity=false com=#SYNCHRONOUS	connection=true inclusion=false similarity=false com=#ASYNCHRONOUS

La *spécialisation* sera utilisée pour créer des styles d'évolutions plus spécifiques et mieux adaptés à certaines classes de problèmes d'évolution. La *composition* sera utilisée pour créer des styles

composites, par combinaison de styles d'évolution existants. Enfin, l'*utilisation* sera utilisée pour faire collaborer des styles d'évolutions.

Ces trois relations ont des éléments relationnels et des axiomes mathématiques (Tableau 3) :

Tableau 3 : Eléments relationnels et axiomes mathématiques des relations en termes de réflexivité (R), transitivité (T) et symétrie (S).

Nom	*Signification*	*Éléments relationnels*	R	T	S
Spécialisation	« est-une-sorte-de »	{Connexion, Similarité, Inclusion}	√	√	–
Composition	« est-composé-de »	{Connexion, Inclusion}	√	√	–
Utilisation	« utilise »	{Connexion}	–	–	–

On considère que plus (respectivement moins) une relation peut être décrite par des éléments relationnels, plus elle offre une sémantique riche (respectivement faible).

- Bibliothèque (*Library*) : similaire au concept d'API (Application Programming Interface), elle est l'unité structurante de l'ensemble des styles d'évolution. Elle constitue le point d'entrée unique pour l'accès aux styles d'évolutions.

b. Représentation des styles d'évolution

Le formalisme choisi doit permettre de distinguer l'entête de la compétence d'un style d'évolution ainsi que les éléments évolutifs concernés. En outre, l'architecte doit pouvoir hiérarchiser les entêtes, les compétences et les éléments évolutifs et disposer de plusieurs vues sur les styles. Le modèle en Y (MY) [Smeda& 08]. Ce modèle a pour socle un tryptique représentant trois aspects importants : le domaine, l'entête et la compétence. Ces aspects représentent respectivement les informations liées aux éléments évolutifs d'une architecture, aux opérations d'évolution sur ces éléments et aux méthodes utilisées pour achever ces opérations. Les bibliothèques de composants réutilisables ont été décrites dans différentes approches de l'ingénierie de la connaissance (*Knowledge Engineering*) [Motta 00]. Nous nous en inspirons pour décrire les styles d'évolution, les structurer et les classer, et les trouver et les utiliser.

MY : le méta-modèle en Y : chaque branche de la lettre Y décrit un concept du style d'évolution, comme le montre la Figure 15. Le centre de l'architecture en Y décrit un entête primitif lié à une compétence élémentaire et décrits dans la terminologie du domaine. De plus, afin de garder la description des trois concepts (domaine, entête, compétence) indépendants les uns des autres, nous explicitons deux relations : lien *inter-concepts* et lien *intra-concepts*. La relation *inter* décrit le lien entre deux concepts différents (par exemple entre un entête et une compétence), alors que la relation *intra* décrit le lien entre deux concepts du même type (par exemple entre un domaine et ses sous-domaines). Ces relations sont utilisées dans le MY comme un outil d'adaptation et d'assemblage pour décrire des styles d'évolution

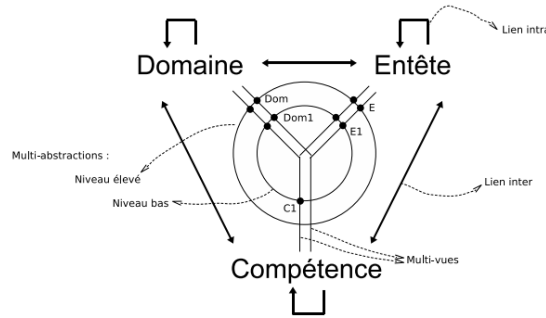


Figure 15: concepts du style d'évolution en utilisant le modèle en Y.

En utilisant le modèle MY, un style d'évolution peut ainsi être décrit par trois aspects : *domaine* (connaissance disponible/éléments évolutifs), *entête* (opérations réalisables sur la base du domaine) et *compétence* (méthode choisie pour réaliser l'opération).

c. Conclusion

L'idée clé de SAEM est d'attacher à chaque élément architectural un ensemble de styles d'évolution. SAEM permet l'*extensibilité* (ajout de style), l'*évolutivité* (la compétence d'un style peut être modifiée sans impacter son utilisation), la *composabilité* (des styles composites peuvent être créés), la *remplaçabilité* (un style peut substituer un autre style ayant la même entête). Il est important de noter que la composition et l'utilisation de styles d'évolution par un autre ne préjuge d'aucune contrainte de précédence dans l'exécution de ces différents styles. Elle n'en précise pas l'ordre d'application et n'a pas vocation à modéliser la dynamique de l'exécution. Elle n'implique pas d'avantage que lors d'une exécution particulière, les différents styles seront systématiquement lancés.

3.5. Bibliothèques pour les styles d'évolution

Nous proposons des bibliothèques pour contenir les styles d'évolution et les relations entre eux, telles que la *spécialisation*, la *composition* et l'*utilisation*, et instanciés sur l'architecture d'un système (Figure 16) :

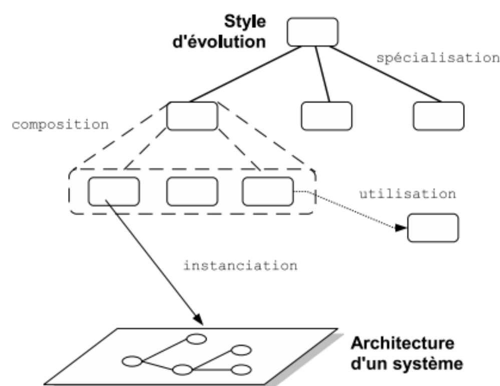


Figure 16 : Organisation de styles d'évolution.

a. Niveaux de modélisation des bibliothèques

Une démarche d'élaboration de bibliothèques pour les styles d'évolution est proposée. Le modèle MY unifie les concepts du paradigme des styles d'évolution en supportant trois niveaux de modélisation : méta-style d'évolution, type de style d'évolution, et instance de style d'évolution. Ce

modèle améliore la réutilisation des styles d'évolutions en supportant une bibliothèque multi-hiérarchique pour les trois niveaux de modélisation. Une infrastructure de réutilisation est bâtie spécifiquement au-dessus de la bibliothèque de niveau type afin d'en contrôler le peuplement et l'interrogation par les architectes, selon que l'on se place du point de vue du fournisseur ou du consommateur d'évolutions.

Notre approche permet d'identifier les éléments réutilisables pour chaque niveau de modélisation, de manière uniforme en utilisant un MY. Ces éléments sont alors stockés dans une bibliothèque liée à chaque niveau de modélisation et utilisée par son niveau inférieur. Par exemple, les éléments réutilisables du niveau méta-style sont : *domaine*, *entête* et *compétence*. Ils sont stockés dans une bibliothèque de niveau E2 et utilisés au niveau E1 (Figure 17).

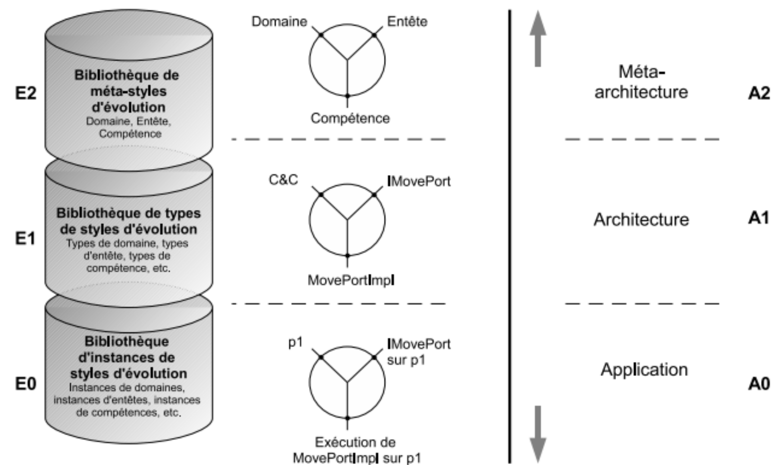


Figure 17 : Les bibliothèques pour les styles d'évolution.

b. Les acteurs

Nous avons identifié trois rôles d'acteurs pour le modèle en Y, chacun associé à un niveau donné (E2, E1 ou E0). Il y a également un quatrième acteur pour gérer les bibliothèques :

- *Le constructeur d'infrastructure d'évolution (CIE) :* définit les concepts de base sur lesquels l'architecte d'application évolutive s'appuie pour établir un ensemble de types de styles d'évolution. Le CIE est concerné par le niveau de modélisation E2.
- *L'architecte d'application évolutive (AAE) :* son rôle est de concevoir et de décrire les types de styles d'évolution qui seront utilisés pour le développement d'application évolutives. Il le fait à travers l'instanciation des concepts fournit par le CIE. L'AAE est concerné par le niveau de modélisation E1.
- *Le développeur d'application évolutive (DAE) :* son rôle est de faire évoluer une architecture logicielle en instanciant le niveau E1. Le DAE est concerné par le niveau E0.
- *Le bibliothécaire de l'évolution (BE) :* construit et gère toutes les bibliothèques. Le bibliothécaire de l'évolution est concerné par tous les niveaux de modélisation. Les bibliothèques sont donc hiérarchisées suivant les trois niveaux de modélisation et se composent :
  - Niveau E2 : des concepts de domaine, d'entête et de compétence définies par le CIE,
  - Niveau E1 : des types de styles d'évolution, créés par assemblage de types de domaines, de types d'entêtes et de types de compétences définies par l'AAE,

- Niveau E0 : des instances de styles d'évolution définies par le DAE, créées à partir d'instances de domaines, d'instances d'entêtes et d'instances de compétences.

c. *Élaboration d'une bibliothèque de styles d'évolution*

L'élaboration d'une bibliothèque de styles d'évolution cible des problèmes d'organisation hiérarchique des styles d'évolutions d'une part, et de définition des besoins de recherche, d'accès, d'adaptation et d'instanciation des styles d'évolutions, d'autre part. Elle produit une infrastructure pour le système de réutilisation, en rendant opérationnels les opérations d'intégration et de recherche des styles d'évolutions dans la bibliothèque de niveau E1. L'accent a été logiquement mis sur la construction de la bibliothèque de niveau E1, là où sont stockés les types de styles d'évolution.

- *Classification de styles d'évolution :* le contenu de la bibliothèque de niveau E1 est défini comme un graphe de styles d'évolution. Ce graphe peut être filtré à travers le point de vue de la spécialisation et de la composition, grâce à la nature hiérarchique des relations sémantiques de spécialisation et de composition. Le filtrage à travers ces deux points de vue isole deux hiérarchies distinctes l'une de l'autre (Figure 18) :

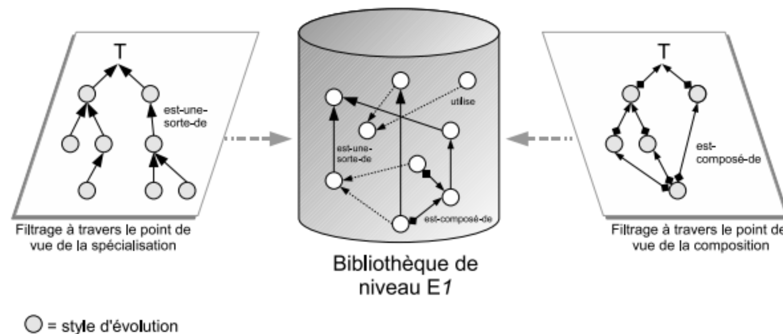


Figure 18 : Filtrage par points de vue sur la bibliothèque de niveau E1.

- *Opération d'intégration dans une bibliothèque* : est basée sur un raisonnement classificatoire qui préserve la structuration de la bibliothèque de niveau E1 selon ses deux points de vue : spécialisation et composition.
- *Opération de recherche dans une bibliothèque* : l'objectif est d'explorer la bibliothèque en vue de réutiliser, autant que possible, les styles d'évolution qui répondent aux besoins exprimés par un architecte d'application évolutive (AAE) ou un développeur d'application évolutive (DAE). La requête doit être exprimée selon le formalisme d'un style d'évolution. Elle est représentée sous forme d'un style abstrait, c'est-à-dire qui possède un entête (donc le problème), mais pas de compétence (car une compétence décrit une solution). Le résultat est l'ensemble des compétences stockées dans la bibliothèque.

3.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Encadrement doctoral (confère *section 7. Détails de l'encadrement doctoral* du CV détaillé) :

2005-2009 : thèse de Olivier LE GOAER (Doctorat en informatique de l'Université de Nantes), « SEAM : Spécification et capitalisation d'expertise de mise à jour dans les architectures logicielles à base de composants : une approche par les styles d'évolution. »

- Taux d'encadrement : 40%.

3.7. Brève synthèse des travaux fondateurs

Les ADLs fournissent un moyen de spécifier un assemblage de primitives d'évolution. Ces spécifications peuvent être contenues à l'intérieur de l'ADL (partie orchestrateur, partie configurateur, etc.) ou à l'extérieur de l'ADL (scripts, transformations, etc.). Très peu d'ADLs permettent de considérer de telles spécifications comme de nouveaux opérateurs d'évolution. En d'autres termes, les seules évolutions qu'il est possible de réutiliser sont de fine granularité.
Les ADLs de première génération, notamment Wright et Darwin, ne se sont pas préoccupés de la problématique de l'évolution au moment de la spécification de l'architecture logicielle. La majorité des ADLs de seconde génération réifient et considèrent comme entités de première classe tous les concepts de base d'une description d'architecture logicielle : composant, connecteur, interface et configuration. Toutefois, la mécanique opératoire offerte par les modèles d'évolution présentés est basique. Les approches sont avares en termes de mécanismes pour réutiliser les opérateurs d'évolution (e.g., extension et composition). Spécifiquement, les propriétés d'évolutivité et de remplaçabilité ne sont pas favorisées. En outre, les approches sont souvent ad-hoc. Si les besoins changent, il faut fournir de nouveaux outils.

3.8. Positionnement et bilan

La principale contribution est le concept de *style d'évolution* qui permet de re-examiner l'évolution comme une activité pouvant être guidée et contrôlée par le respect d'archétypes prédéfinis. Ce concept est le concept au cœur du méta-modèle SEAM pour la spécification et la capitalisation de savoir-faire d'évolutions. L'objectif est la proposition d'une approche pour la détection, la spécification, la capitalisation et la réutilisation du savoir-faire et de l'expertise de toute activité d'évolution effectuée. Il est à noter que parallèlement à nos travaux, Garlan et son équipe ont proposé un concept de *Style d'évolution*, et ils se positionnent par rapport à nos travaux [Garlan 08] [Garlan& 09] [Barnes& 14]. Les *styles d'évolution* proposés par Garlan et son équipe sont des chemins d'évolution, avec pour objectif de permettre de choisir le chemin optimal pour réaliser les objectifs métier d'une organisation [Barnes& 14].

SEAM est un cadre conceptuel technologiquement neutre où différents acteurs œuvrent collectivement en vue de capitaliser leur savoir-faire d'évolution, de l'enrichir et de l'exploiter tel quel ou pour en définir de nouveaux. Tout style d'évolution explicite une action d'évolution possible. Il est forcément attaché à une entité pouvant évoluer. Dans le cadre des approches à base de composants, un style d'évolution sera attaché à un composant, une interface, un connecteur ou une configuration. Nous considérons qu'un style d'évolution n'est pas une unité monolithique mais une agrégation de trois composantes indissociables :

1. *Domaine* : il s'agit d'un domaine de connaissance définit par un ensemble de concepts et de relations entre eux, tels que le domaine orienté-objet ou un système bancaire. Un savoir-faire d'évolution est dépendant d'un domaine et ne peut pas s'en affranchir.
2. *Entête* : il s'agit d'une signature ou d'un contrat stipulé par le savoir-faire : entrées/sorties et assertions. En d'autres termes, cela explicite le système logiciel de départ (avant évolution) et celui d'arrivée (après évolution).
3. *Compétence* : il s'agit de l'expression du savoir-faire, généralement du code à exécuter. La compétence s'engage à remplir le contrat fixé par l'entête.

La technique de méta-modélisation en Y offre un support de description simple et minimal pour le paradigme des styles d'évolution. Cette proposition peut être rapprochée des travaux entrepris dans

UPML [Fensel& 03] (*Unified Problemsolving Method description Language*) en particulier, où l'objectif reste d'obtenir une application de système à base de connaissance à partir de blocs réutilisables.

Une autre contribution est l'organisation des styles d'évolution au travers de trois types de relations : l'*utilisation* permettant à un style d'évolution d'invoquer un autre nécessaire à son exécution, la *composition* de styles pour définir des styles de granularité plus importante et la *spécialisation* qui permet de raffiner des styles existants pour adapter leur définition à des contextes plus ciblés. Ces relations, notamment les deux dernières, permettent d'organiser les styles d'évolution au sein de librairies. L'exploitation de ces librairies se base sur le même algorithme, aussi bien pour rechercher un style d'évolution adapté à un besoin d'évolution que pour insérer un style d'évolution nouvellement créé.

Validation : la validation du prototype de SEAM se fait dans le cadre du projet ZOOM (confère section 6.2.6 du CV détaillé). L'objectif du projet ZOOM était de décrire des architectures logicielles dédiées aux réseaux de conditionnement d'air, gaines de ventilation machine et aux réseaux de tuyauterie dans la fabrication navale. La validation de SAEM concerne principalement l'expression des savoir-faire d'évolution associés à la planification de câbles, gaines et tuyaux associés aux métiers de construction navale et aux évolutions (réelles ou simulées) de ces plans.

Ma contribution : mon implication dans SEAM est le co-encadrement de la thèse à hauteur de 40%, avec Mourad Oussalah (30%) et Djamel Seriai (30%).

Valorisation : ces travaux ont donné lieu à 15 publications, notamment internationales dont une conférence de rang A, IEEE COMPSAC 2008 (référence [30] du CV détaillé) et 7 conférences de rang B, SEKE, SEDE, SERP, IEEE IRI, ICSOFT (références [43, 44, 45, 48, 49, 50, 53] du CV détaillé).

4. Extraction d'architectures logicielles d'un système orienté objet : une approche par exploration

4.1. Fiche d'identité

Titre :	ROMANTIC : Extraction d'architectures logicielles d'un système orienté objet.
Paradigme(s) :	Architectures logicielles à base de composants – ADLs – Objets
Principales contributions :	. Modèles et critères de la sémantique et la qualité architecturale pour l'approche ROMANTIC pour extraire une architecture à base de composants à partir d'un système orienté objet existant. Le processus quasi-automatique d'identification d'architectures est formulé comme un problème d'optimisation et en le résolvant au moyen de méta-heuristiques. . L'exploration s'appuie sur la sémantique et la qualité architecturale et l'architecture intentionnelle, la documentation et des recommandations de l'architecte pour sélectionner les meilleures solutions.
Mots-clés	Architecture logicielle, composant logiciel, connecteur, extraction, sémantique architecturale, qualité, architecture intentionnelle, méta-heuristiques.
Collaborateur(s) :	S. Chardigny (doctorant), M. Oussalah (directeur de thèse, Université de Nantes), D. Seriai (co-encadrant, Ecole des Mines de Douai).
Mon implication :	Co-encadrement à 30% de la Thèse de S. Chardigny.
Cadre de collaboration :	Bourse Ecole des Mines de Douai.
Période :	2006-2009.
Principales publications :	1 chapitre d'ouvrage ; 1 revue nationale : l'Objet 2008 ; 1 conférence de rang A : IEEE WICSA 2008 ; 2 conférences de rang B : IEEE CSMR 2008, ECSA 2008 ; 1 symposium international et 2 conférences/ateliers nationaux.

Figure 19 : Fiche synthétique.

4.2. Synthèse

Les travaux concernent l'extraction de l'architecture logicielle de systèmes objet existants. L'objectif est d'extraire une architecture adaptée et adéquate d'un système existant afin de mieux piloter, à terme, ses évolutions.

Pour beaucoup de systèmes existants, aucune représentation fiable de leur architecture logicielle n'est disponible. Afin de pallier à ce manque, source de nombreuses difficultés principalement lors des phases de maintenance et d'évolution, nous proposons dans ces travaux une approche, appelée ROMANTIC (*Re-engineering of Object-oriented sytesMs by Architecture extractioN and migraTIon to Component based ones*), visant à extraire une architecture à base de composants à partir d'un système orienté objet existant. L'idée première de cette approche est de proposer un processus quasi-automatique d'identification d'architectures en formulant le problème comme un problème d'optimisation et en le résolvant au moyen de méta-heuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système en utilisant la sémantique et la qualité

architecturale pour sélectionner les meilleures solutions. Le processus s'appuie également sur l'architecture intentionnelle du système, à travers l'utilisation de la documentation et des recommandations de l'architecte.

Pour obtenir cette représentation, notre processus doit sélectionner parmi l'ensemble des représentations possibles d'un système, la meilleure possible par rapport à un ensemble de propriétés de qualité et en respectant un ensemble de guides tel que la documentation du système. Ainsi, nous proposons de formuler ces propriétés comme des contraintes mesurables et nous utilisons une approche d'optimisation métaheuristique afin d'extraire l'architecture qui satisfait le maximum de ces contraintes. Notre choix d'une approche d'optimisation métaheuristique est motivé par les travaux dans le domaine du *Search-based software engineering* (SBSE) qui ont montré l'efficacité de ces techniques pour résoudre des problèmes de ce type [Harman 07]. Nous modélisons donc le problème d'extraction comme un problème d'optimisation en définissant les différents éléments nécessaires à la résolution de ce type de problème.

4.3. Verrous scientifiques

Le principal verrou consiste à identifier au sein du code objet existant les éléments qui peuvent être représentés par un élément architectural à partir des éléments de première granularité présentes dans le code (classes, interfaces et relations). La difficulté est d'arriver à exploiter de manière semi-automatique l'ensemble des informations disponibles sur un système existant. Si l'analyse du code peut être faite, il reste compliqué d'identifier et de projeter des portions de code au sein de composants, de connecteurs et d'interfaces. D'autres sources d'information importantes sont la documentation et l'architecture intentionnelle. Cette dernière représente les intentions et les réflexions des concepteurs du système ou des architectes chargés de sa maintenance et illustrent un système légèrement différent de l'implémentation existante. Par sa nature, cette architecture intentionnelle est très précieuse pour réaliser l'extraction de l'architecture logicielle d'un système mais elle est complexe à obtenir. Les travaux d'extraction existant ne permettent pas de concilier l'utilisation de l'architecture intentionnelle avec un bon niveau d'automatisation.

Le second verrou consiste à analyser l'ensemble des architectures logicielles potentielles et d'identifier automatiquement la plus adaptée au code existant.

Les contributions sont présentées ci-après :

4.4. La démarche ROMANTIC pour l'extraction.

Le domaine de l'extraction d'architecture (*Software Architecture Recovery, SAR*) vise à apporter une représentation de l'architecture réelle d'un système. L'extraction d'architecture d'un système orienté objet consiste à établir une correspondance entre son code source et une abstraction de ce système en termes d'éléments architecturaux : les composants qui décrivent la partie métier, les connecteurs qui décrivent les interactions et la configuration qui représente la topologie des connexions entre les composants [Riva 00] [Krikhaar 97][Jazayeri& 00].

Le processus de ROMANTIC permet de sélectionner parmi l'ensemble des architectures pouvant être abstraites du système, la meilleure possible par rapport à un ensemble de propriétés de qualité et en respectant un ensemble de guides tel que la documentation du système. Ainsi, nous proposons de formuler ces propriétés comme des contraintes mesurables et nous utilisons une approche d'optimisation méta-heuristique afin d'extraire l'architecture qui satisfait le maximum de ces

contraintes. L'approche d'extraction est *interactive* (interactions avec l'utilisateur quand nécessaire), *itérative* et *semi-automatique* (utilisant des méta-heuristiques pour explorer l'espace des solutions du problème d'extraction).

a. Un modèle de correspondance Objet/Architecture

La modélisation du problème d'extraction consiste essentiellement à définir l'espace des solutions qui doit être exploré. Nous avons donc proposé un modèle de correspondance entre les entités objets et architecturales (COA) dont les instances constituent les éléments de cet espace. Ce modèle COA introduit les notions de *contours d'éléments architecturaux* qui constituent le pont entre le paradigme objet et celui architectural. Les contours mettent en correspondance les composants avec un ensemble de classes et les connecteurs avec un ensemble de méthodes. Le modèle COA est également la base du traitement équivalent apporté aux composants et aux connecteurs. En effet, le modèle considère ces deux éléments architecturaux de manière identique et définit deux entités similaires pour représenter les correspondances de ces éléments avec les éléments objets : les contours de composants et ceux de connecteurs.

Le modèle d'extraction que nous proposons repose sur quatre niveaux conceptuels :

1. *Le niveau du code source :* ce niveau est celui du système logiciel analysé. Il contient le code source dans sa représentation textuelle.
2. *Le niveau modèle du code source* : ce niveau contient un modèle du code source. Les entités de ce modèle sont des éléments du code source et dépendent de la granularité de l'approche. Ce modèle doit permettre une analyse syntaxique et sémantique du code source.
3. *Le niveau de mise en correspondance :* ce niveau contient des regroupements des entités du niveau modèle du code source. Ce niveau établit le lien entre les éléments architecturaux et les entités du code source.
4. *Le niveau architectural :* ce niveau est celui de l'architecture du système. Il contient les éléments architecturaux, composants et connecteurs, qui composent le système représenté par le niveau du code source.

Sur les deux premiers niveaux, notre processus considère un système à objets avec les concepts classiques associés. Sur le niveau architectural, le processus propose d'extraire une architecture logicielle à base de composants conforme aux définitions. Cette architecture doit donc représenter de manière explicite les trois éléments architecturaux : composants, connecteurs et configuration. Vient ensuite le niveau de mise en correspondance entre les entités du code source et architecturales. Chaque entité du modèle de correspondance associe un type d'élément architectural et un regroupement d'entité du modèle du code source.

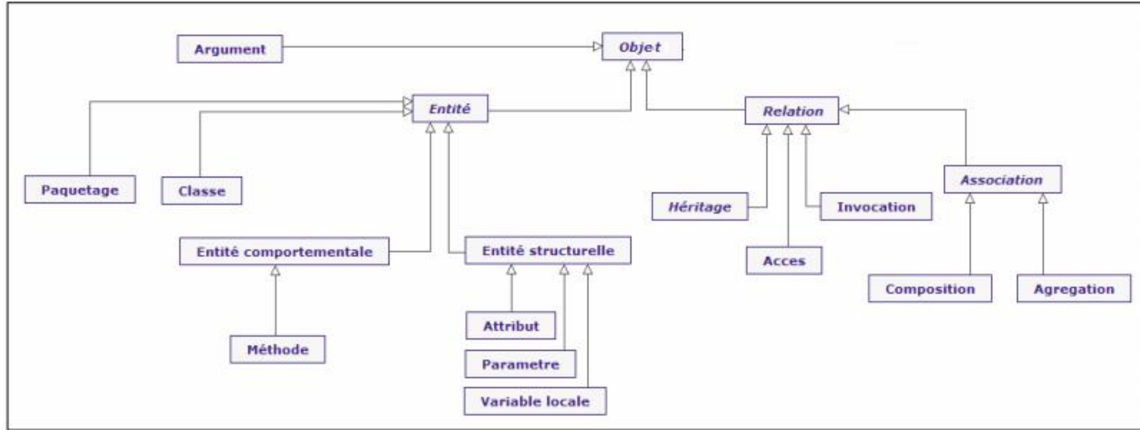


Figure 20 : Modèle du code source.

Les éléments abstraits *Objet*, *Entité*, et *Relation* représentent des super-types permettant au modèle d'être extensible. Ainsi, *Objet* est le super-type racine. *Entité* est le super-type de toute entité structurelle et comportementale d'un programme objet (classe, paquetage, méthode…). *Relation* est le super-type de tout type de lien (association, héritage, composition, agrégation…).

b. Modèle de mise en correspondance

Il permet une mise en correspondance des entités objets avec les éléments architecturaux : les composants pour la partie métier, les connecteurs pour les interactions et la configuration pour la topologie des composants et de leurs connexions. Nous proposons notre modèle de correspondance en définissant une architecture comme une partition des classes du système. Nous appelons contour chaque élément de cette partition. Ces contours représentent chacun un composant. Ils contiennent des classes qui peuvent appartenir à différents paquetages (cf. Figure 21-(a)). Nous assimilons l'ensemble des interfaces du composant à « l'interface du contour ». Chaque contour est constitué de deux ensembles de classes : « l'interface du contour » qui contient l'ensemble des classes qui ont un lien avec des classes situées à l'extérieur du contour, par exemple un appel de méthode vers l'extérieur ; et le « centre » qui contient le reste des classes du contour. Comme le montre la Figure 21-(b), nous assimilons l'ensemble des interfaces du composant à l'interface du contour et le composant au contour. Nous nous concentrons ici sur l'extraction de composants. Pour cela nous considérons que les connecteurs sont tous les liens existants entre deux composants.

- *Espace de recherche* : l'ensemble des instances de ce modèle représente l'ensemble des architectures logicielles pouvant abstraire le système et constitue donc l'espace de recherche de notre processus d'optimisation. Ainsi, l'espace de recherche de notre problème d'optimisation est composé de toutes les partitions des classes du système.

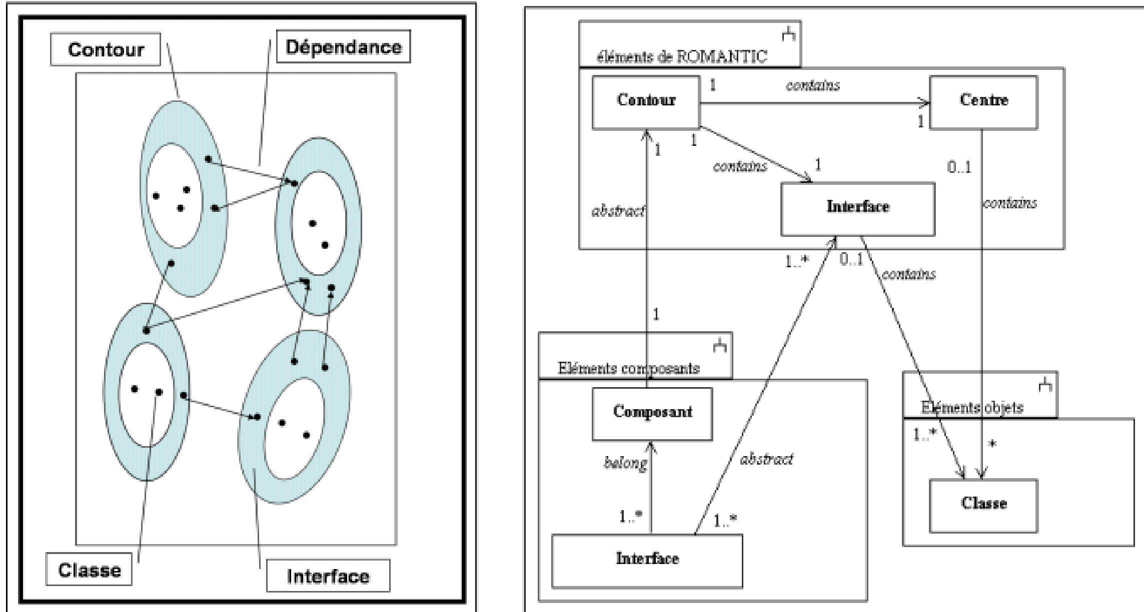


Figure 21 : (a) Eléments de contours – (b) Modèle de correspondance.

c. *Guider le processus d'extraction :*

En plus de la définition de l'espace de recherche, nous avons besoin de définir des guides pour diriger le processus d'exploration de l'espace, en nous basons sur plusieurs sources d'informations (cf. Figure 22).

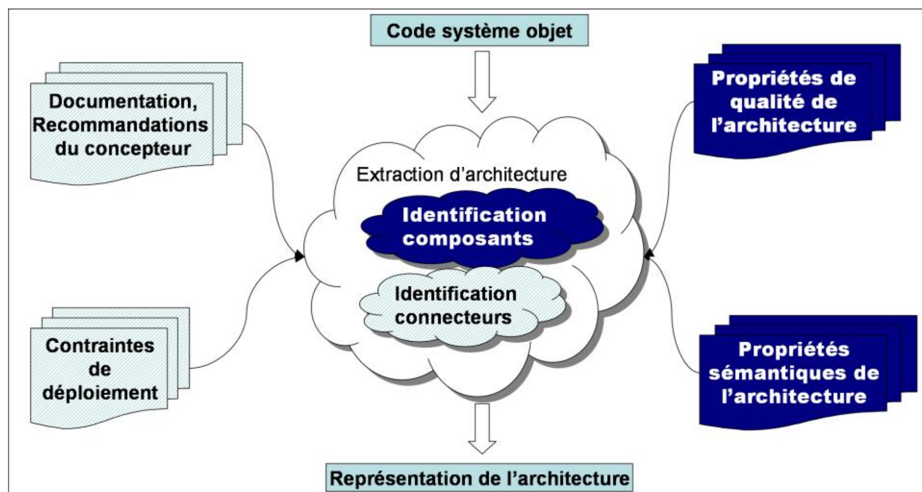


Figure 22 : Guides de l'extraction.

Ces sources d'information sont *les propriétés de qualité de l'architecture* et les *propriétés de sémantique de l'architecture*. Ces deux propriétés permettent de définir la fonction objectif. Il faut définir une mesure de la validité sémantique et une mesure des caractéristiques de qualité d'une architecture. Quant à la *documentation et recommandations du concepteur* ainsi que les *contraintes de déploiement*, elles permettent de fournir des informations pour réduire l'espace de recherche des solutions.

d. *Qualité d'une architecture*

La définition de la fonction objectif nécessite une définition précise de ce qu'est une architecture sémantiquement valide. Nous considérons qu'une architecture est sémantiquement valide si ses éléments (composants, connecteurs et configuration) sont sémantiquement valides. Ainsi, la validité sémantique d'une architecture s'appuie sur la validité sémantique de ses éléments. Illustrons notre approche sur le concept de Composant, l'approche étant identiquement appliquée aux connecteurs et configurations).

Sémantiquement, nous considérons qu'*un composant est un élément logiciel qui (a) est composable sans modification, (b) peut être distribué de manière autonome, (c) encapsule une fonctionnalité,* et *(d) qui adhère à un modèle de composant* [Jacobson& 97], [Szyperski 98], [Heinemann 01], [Luer& 02]. Un composant a ainsi trois caractéristiques sémantiques : la *composabilité*, *l'autonomie* et la *spécificité*. La spécificité impose qu'un composant ait un nombre limité de fonctionnalités.

Les propriétés des composants ne peuvent pas être mesurées sur les contours. Selon notre modèle d'évaluation (Figure 23-B), nous relions les propriétés des composants aux propriétés mesurables des contours. La cohésion moyenne des classes de l'interface est une mesure de la cohésion des services par interface. La cohésion des interfaces, la cohésion interne du composant et le couplage interne du composant peuvent être respectivement mesurés par les propriétés du contour *cohésion de l'interface*, *cohésion du contour* et *couplage interne du contour*. Pour relier le nombre d'interfaces fournies à une propriété des contours, nous associons une interface fournie du composant à chaque classe de l'interface du contour qui possède une méthode publique. Grâce à cette hypothèse, nous pouvons mesurer le nombre d'interfaces fournies en utilisant la propriété *nombre de classes de l'interface possédant une méthode publique*. Le nombre d'interfaces requises peut être évalué par le couplage entre le composant et l'extérieur. Ce couplage est relié au couplage externe du contour. Ainsi, nous mesurons cette propriété par la propriété *couplage externe du contour*.

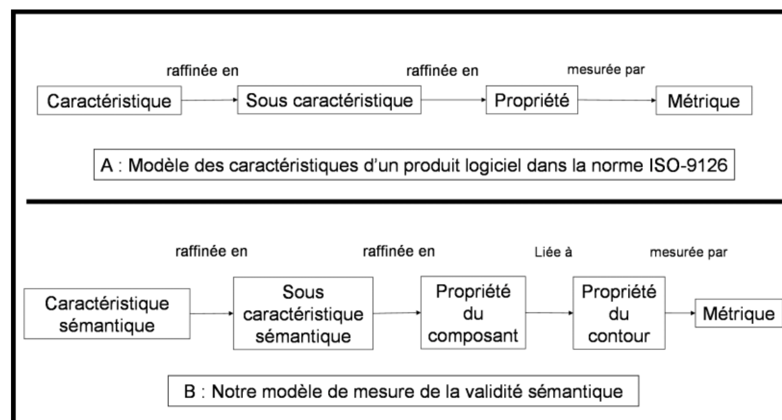


Figure 23 : Modèle de mesure de caractéristiques dans la norme ISO-9126 et dans notre approche.

Après avoir établi les liens entre les sous-caractéristiques sémantiques et les propriétés des contours (Figure 24), nous définissons les métriques :

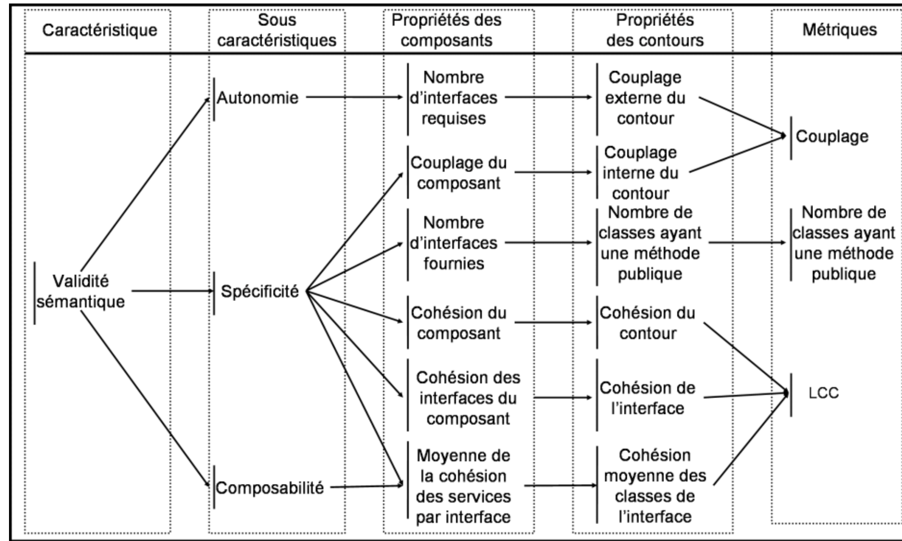


Figure 24 : Notre modèle de mesure de la validité sémantique des composants.

- La fonction objectif permet de sélectionner la meilleure au vu de la validité sémantique et la qualité de l'architecture :
- *La validité sémantique de l'architecture :* se base sur la validité sémantique de ses composants et ses connecteurs, Sem_{con} et Sem_{comp}. Pour chacun d'entre eux, il faut évaluer leur spécificité, autonomie et composabilité ainsi que leur sémantique. La *validité sémantique de l'architecture* est une moyenne pondérée de la validité sémantique de chaque élément de l'architecture extraite. Ainsi, nous définissons la fonction d'évaluation d'une solution *s* selon l'équation suivante, où $E_{comp}(s)$ et $E_{con}(s)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de *s* :

$$Sem\,(s) = (\lambda_6 + \lambda_7 + |E_{comp}\,(s)| + |E_{con}\,(s)|)^{-1} \cdot \left(\lambda_6 \cdot \sum_{c \in E_{comp}(s)} Sem_{comp}\,(c) + \lambda_7 \cdot \sum_{c \in E_{con}(s)} Sem_{con}\,(c)\right)$$

  Les pondérations sont fixées par l'architecte et permettent de définir l'importance relative accordée aux composants et aux connecteurs.
- *La qualité architecturale* : sa mesure se base sur celle de sa maintenabilité $M_{aint}(s)$ et sa fiabilité $F_{iab}(s)$. Ainsi, nous définissons la fonction d'évaluation d'une solution *s* selon l'équation suivante, où $E_{comp}(s)$ et $E_{con}(s)$ désignent respectivement l'ensemble des contours de composants et de connecteurs de *s* :

$$Qual\,(s) = \frac{1}{\sum_i \lambda_i} (\lambda_8 \cdot Maint\,(s) + \lambda_9 \cdot Fiab\,(s))$$

  Le poids associé à chaque fonction autorise l'architecte à modifier l'importance de chaque caractéristique de qualité.
- *La fonction objectif du processus d'exploration :* permet de mesurer la qualité d'une solution et de comparer les solutions entre elles tout au long du processus d'extraction. Elle permet d'évaluer la qualité d'une solution en mesurant la validité sémantique et la qualité architecturale de l'architecture correspondante à la solution.

$$Goal\,(s) = \frac{1}{\sum_i \lambda_i} (\lambda_8 \cdot Sem\,(s) + \lambda_9 \cdot Qual\,(s))$$

e. Processus d'extraction des informations intentionnelles

Les informations utilisées pour définir la fonction objectif reposent uniquement sur le code source de l'application. Cette approche a ses limites. Nous utilisons également les informations intentionnelles pour prendre en compte l'architecture telle qu'imaginée par les concepteurs, en nous basons sur deux sources d'informations intentionnelles : la documentation et les recommandations de l'architecte. Elles représentent des guides auxiliaires de l'extraction. Elles permettent d'étendre les sources d'informations utilisées par le processus d'extraction au-delà du code source.

Nous utilisons un processus automatique d'extraction des informations intentionnelles contenues dans la documentation issue de chaque phase majeure du cycle de vie : la conception avec les diagrammes UML, l'implémentation avec les informations informelles contenues dans le code source et la maintenance avec les informations extraites des outils de versionnement. Nous récupérons par la suite les recommandations de l'architecte en termes d'informations intentionnelles et contextuelles. Ces informations sont précieuses pour réduire l'espace des recherches. Cette réduction se fait de deux manières. La première utilise les informations pour éliminer certains éléments de l'espace de solutions qui sont en contradiction avec l'architecture intentionnelle. La seconde utilise les informations extraites pour cibler certaines zones de l'espace des solutions qui doivent contenir des solutions plus proches de l'architecture intentionnelle.

L'ensemble du processus d'extraction de l'architecture intentionnelle et des contraintes contextuelles est conçu pour être « à la disposition » des experts. En effet, si un architecte est disponible, il peut grandement influencer les éléments extraits de la documentation et donc agir sur le processus d'extraction d'architecture utilisant ces guides. Dans le cas où l'architecte est peu ou pas disponible, il peut se reposer sur des mécanismes automatiques qui, s'ils sont moins efficaces, permettent d'utiliser les informations intentionnelles malgré le manque de disponibilité des experts.

4.5. Mise en pratique

La mise en pratique de l'approche ROMANTIC s'est faite sur des méta-heuristiques. Elles peuvent être classées selon trois critères : la complexité algorithmique, la complexité de la mise en œuvre et la capacité à éviter les minima locaux. L'objectif est d'avoir un algorithme peu complexe algorithmiquement et en termes de mise en œuvre et qui puisse éviter les minima locaux. Nous avons utilisé les méta-heuristiques du recuit simulé et de l'algorithme génétique. Ces méta-heuristiques sont utilisées pour une exploration de l'espace des architectures possibles. La fonction objectif est basée sur la sémantique des composants.

a. Algorithmes pour l'exploration

Cette sous-section présente comment les éléments clés des méta-heuristiques ont été projetés sur nos travaux, sans rentrer dans le détails des spécifications :

1. *Recuit simulé :* la métaheuristique du recuit simulé (*«Low-Temperature Simulated Annealing»*) [Laarhoven& 87] consiste essentiellement en une série de tentatives de modification sur une solution d'un problème d'optimisation. Les changements qui accroissent la qualité de la solution sont acceptés et la solution modifiée devient le point de départ d'une nouvelle série de tentatives de modification. Comme la plupart des métaheuristiques, cet algorithme nécessite une fonction objectif, une représentation du problème et un moyen d'altérer les solutions. En plus, il nécessite un schéma de refroidissement (cooling schedule) qui détermine à quelle vitesse le paramètre de température

décroît et ainsi la vitesse d'exécution et la qualité de la solution obtenue. Enfin, le recuit nécessite une solution initiale qui est le point de départ de l'optimisation.
Outre la fonction objectif, nous définissons trois opérateurs de manipulation pour le schéma de refroidissement : *move(c, s1, s2)* qui déplace une classe c du contour s1 vers le contour s2 ; *extract(c, s)* qui déplace une classe c du contour s vers un nouveau contour ; *fusion(s1, s2)* qui regroupe les contours s1 et s2 en un nouveau contour contenant toutes les classes de s1 et de s2. Pour la solution de départ, nous avons fait le choix de prendre comme solution de départ les composantes fortement connexes, et donc de commencer par l'identification des composants réutilisables par la localisation des sous-ensembles connexes parmi les éléments du système. Ces composantes fortement connexes forment une partition des sommets du graphe et par conséquent une partition des classes du système. Malgré une facilité de mise en œuvre et une consommation raisonnable de l'espace mémoire, le recuit simulé reste une solution qui nécessite des temps de calcul importants.

2. *Algorithmes Génétiques* : consistent essentiellement à manipuler une population de solutions, représentées par des chromosomes, au moyen d'un ensemble d'opérateurs dit génétiques [Syswerda 89]. A chaque génération, l'algorithme sélectionne un ensemble d'individus, opère des croisements et/ou des mutations pour obtenir une nouvelle génération et ainsi recommencer le processus. Nous avons choisi d'utiliser trois types de chromosomes : chromosome des composants (chaque gène représente un contour de composants. Il contient l'ensemble des classes qui constituent le contour et l'ensemble des contours d'interface associés à ce contour), des connecteurs (chaque gène représente un contour de connecteur. Il contient l'ensemble des méthodes qui constituent le contour et l'ensemble des méthodes requises par le contour) et de la configuration (chaque autre gène représente un contour de connecteurs utilisé dans l'architecture. Il contient l'ensemble des contours de composant reliés à ce contour). La population initiale contient à la fois des solutions générées de manière aléatoire et des solutions calculées à partir de différentes solutions.
L'algorithme génétique offre en plus la possibilité d'utiliser les résultats du recuit ou du regroupement hiérarchique pour diversifier sa population initiale et augmenter la qualité des résultats. Par contre le méta-modèle spécifique des algorithmes génétiques impose un codage particulier et constitue donc l'inconvénient majeur de ces algorithmes.

b. Cas d'étude

Pour comparer le recours aux deux méta-heuristiques, nous avons appliqué ROMANTIC sur deux systèmes : Jigsaw et ArgoUML. L'étude qualitative menée a mis en évidence l'adéquation entre l'architecture extraite et celle prévue manuellement en utilisant nos connaissances des systèmes. Nous avons ensuite comparé les algorithmes proposés selon deux axes : la vitesse et le rapport vitesse/ précision. Pour cela nous avons mesuré la vitesse à travers le nombre d'évaluation de la fonction objectif et la précision à travers le résultat de notre fonction objectif :

- *Le recuit simulé :* les propriétés du recuit simulé le rendent particulièrement utile dans les cas où le temps de calcul est limité et que la qualité du résultat n'est pas cruciale. En effet, en utilisant le recuit avec une limite du nombre d'évaluation de la fonction objectif et en le redémarrant en cas de dépassement, il est possible d'obtenir rapidement une bonne représentation de l'architecture d'un système ;
- *L'algorithme génétique :* les propriétés de cet algorithme le rendent particulièrement efficace dans les cas où le temps de calcul est secondaire par rapport à la qualité du résultat. Il permet en

effet un réglage fin du temps de calcul et offre la meilleure garantie d'une qualité de résultat en rapport avec le temps machine utilisé.

4.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Encadrement doctoral (confère *section 7. Détails de l'encadrement doctoral* du CV détaillé) :

2006-2009 : thèse de Sylvain CHARDIGNY (Doctorat en informatique de l'Université de Nantes), « Restructuration et rétro-conception dans les architectures logicielles à base de composants »

- Taux d'encadrement : 30%.

4.7. Brève synthèse des travaux fondateurs

Les processus d'extraction ne font pas de l'évolution ni de la maintenance mais ils en sont parfois une étape préalable indispensable pour les systèmes sans description architecturale fiable. Parmi les usages les plus fréquents des résultats de processus d'extraction d'architecture, citons : la re-documentation et la compréhension, la maintenance, la réutilisation, l'analyse, la co-évolution ou encore la conformité [Polet& 07]. L'extraction d'architecture d'un système peut être réalisé par une approche descendante ou montante. L'approche descendante commence par calculer une vue architecturale du système puis par établir les liens entre cette vue et le code source du système [Medvidovic& 06). Cette approche est difficilement automatisable et nécessite des personnes ayant un bon niveau d'expertise. Au contraire, l'approche montante calcule des regroupements dans les entités du code source et propose ensuite des éléments architecturaux en correspondance avec ces regroupements [Tzerpos& 96] [Anquetil& 99].

De nombreux travaux sont présentés dans la littérature pour réaliser l'extraction d'une architecture d'un système orienté objet, tels que [Harris& 95], [Tzerpos& 96], [Krikhaar 97], [Kazman& 98], [Anquetil& 99], [Riva 00], [Jazayeri& 00], [Koschke 00], [Medvidovic& 06], [Polet& 07]. La plupart des travaux sont basés sur l'expertise humaine : certains utilisent l'expertise de l'architecte qui utilise l'approche alors que d'autres utilisent l'expertise de ceux qui ont proposé cette approche. Ceux qui choisissent l'architecture intentionnelle bénéficient de précieuses informations mais nécessitent des experts pour réaliser l'extraction [Kazman& 99].

Dans ROMANTIC, nous utilisons la sémantique architecturale afin de réduire le besoin d'expertise qui peut être coûteux ou indisponible. Cette entrée sera complétée par les autres guides identifiés et en particulier la documentation qui est utilisée dans les travaux existants à travers l'expertise humaine. Les techniques utilisées pour extraire l'architecture sont variées et peuvent être classées selon leur degré d'automatisation. Premièrement, certaines approches sont quasi manuelles. Par exemple, Focus [Medvidovic& 06] est un guide pour un processus hybride qui regroupe les classes et met en correspondance les entités extraites avec une architecture idéalisée à partir du style architectural en fonction de l'expertise humaine. Deuxièmement, la plupart des approches proposent des techniques semi-automatiques. Elles automatisent les aspects répétitifs de l'extraction mais l'expert dirige le processus de raffinement ou d'abstraction menant à l'identification des éléments architecturaux. Ainsi ManSART [Harris& 95] tente de mettre en correspondance les éléments du code source avec le style architectural et les patterns définis par l'expert. Enfin certaines techniques sont quasi-automatiques. On peut par exemple citer les algorithmes de regroupement qui sont utilisés pour produire des regroupements cohésifs et faiblement interconnectés [Anquetil& 99]. ROMANTIC est également quasi-automatique et elle est la seule approche à utiliser une métaheuristique d'exploration. La principale différence avec les autres approches quasi-automatiques est que nous raffinons les définitions couramment admises des composants en un ensemble de caractéristiques sémantiques et

modèles de mesure alors que les autres travaux utilisent l'expertise de leurs auteurs afin de définir les règles qui contrôlent le processus. La différence majeure avec ROMANTIC est que son niveau architectural ne préjuge pas du type des composants utilisés pour décrire le système. Il permet également de s'appuyer aussi bien sur l'analyse du code source que la prise en compte d'autres sources telles que la documentation ou encore l'architecture intentionnelle.

4.8. Positionnement et bilan

Le modèle ROMANTIC (*Re-engineering of Object-oriented sytesMs by Architecture extractioN and migraTIon to Component based ones*) s'appuie sur un processus d'architecturation, basé sur des approches stochastiques pour rechercher la meilleure architecture possible. L'idée première de cette approche est de proposer un processus quasi-automatique d'identification d'architecture en formulant le problème comme un problème d'optimisation et en le résolvant au moyen de méta-heuristiques. Ces dernières explorent l'espace composé des architectures pouvant être abstraites du système en utilisant la sémantique et la qualité architecturale pour sélectionner les meilleures solutions. Le processus s'appuie également sur l'architecture intentionnelle du système, à travers l'utilisation de la documentation et des recommandations de l'architecte.

Validation : nous avons testé le processus d'extraction sur Jigsaw, qui est un serveur Web java et qui contient autour de 300 classes. Le calcul des composantes fortement connexes nous donne une première partition de 100 contours. En faisant plusieurs simulations, les résultats restent relativement similaires et ont pratiquement le même résultat : 87%. Le résultat est donc une architecture à base de composants proche de celle attendue.

Ma contribution : mon implication dans ROMANTIC est le co-encadrement de la thèse à hauteur de 30%, avec Mourad Oussalah (30%) et Djamel Seriai (40%).

Valorisation : ces travaux ont donné lieu à 9 publications, notamment internationales dont une conférence de rang A, IEEE WICSA 2008 (référence [31] du CV détaillé) et deux conférences de rang B, IEEE CSMR 2008 et ECSA 2008 (références [47, 52] du CV détaillé).

5. Spécification de processus de migrations par approche formelle : théorie des graphes

5.1. Fiche d'identité

Titre :	Evolution Patterns: Designing and Reusing Architectural Evolution Knowledge to Introduce Architectural Styles
Paradigme(s) :	Architectures logicielles à base de composants – ADLs
Principales contributions :	. Définition du concept de *patron d'évolution* pour spécifier une évolution éprouvée, automatisée et réutilisable. Le concept a été appliqué sur la restructuration d'une architecture logicielle pour appliquer un style architectural. . Formalisation, exécution et analyse de l'évolution par la théorie des graphes et des règles de transformation de graphes.
Mots-clés :	Evolution logicielle, réutilisation d'un savoir-faire d'évolution, patron, architecture logicielle, style architectural, transformation de graphe, ADLs.
Collaborateur(s) :	T. Mens (Professeur, Université de Mons, Belgique) A. Lansmanne (Master 2, Université de Mons) – co-encadrement à 50%.
Mon implication :	50 % Collaboration.
Cadre de collaboration :	Séjour invité de 6 mois durant mon CRCT 2008-2009.
Période :	2008-2010.
Principales publications :	1 conférence de rang B : IEEE ECBS 2010 ; 2 conférences internationales ; 1 conférence nationale.

Figure 25 : Fiche synthétique.

5.2. Synthèse

Ces travaux s'inscrivent dans les problématiques d'évolution de descriptions d'architectures logicielles formalisées via un ADL. Notre objectif principal est d'offrir une manière disciplinée de réutiliser des processus d'évolution au sein des architectures logicielles. Nous nous concentrons sur un besoin d'évolution spécifique et récurrent : introduire un style architectural en restructurant une architecture logicielle existante. Pour faire face à un besoin d'évolution récurrent, nous introduisons le concept de *patron d'évolution*. Il formalise une évolution architecturale à travers un ensemble de concepts et un processus d'évolution réutilisable. Ainsi, nous proposons de formaliser et d'automatiser les modèles d'évolution avec pour objectif de guider l'évolution des descriptions architecturales. L'approche globale suit trois étapes successives pour spécifier un processus d'évolution réutilisable : (1) réifier les concepts architecturaux qui peuvent évoluer ; (2) spécifier un ensemble minimal d'opérations d'évolution récurrentes sur ces concepts ; et (3) spécifier le processus d'évolution à travers un flux spécifique d'opérations d'évolution appliquées aux concepts architecturaux identifiés. Ce processus est proposé comme un patron d'évolution, donc réutilisable dans des contextes ayant des problématiques similaires.

Comme préconisé par la définition de la norme ISO/CEI 42010[11] ou encore [Medvidovic & 00], un ADL devrait proposer un support à l'évolution des descriptions d'architectures logicielles. Pourtant,

[11] Une architecture logicielle est « l'organisation fondamentale d'un système incarnée dans ses composants, leurs relations [...] et les principes guidant sa conception et son évolution »

contrairement au support formel et automatisé pour les descriptions d'architectures logicielles été largement abordé, leur évolution, pourtant tout aussi cruciales, sont beaucoup moins bien abordées.

Mots clés : évolution logicielle, réutilisation des connaissances d'évolution, modèle, architecture logicielle, style architectural, transformation de graphe, langage de description d'architecture.

5.3. Verrous scientifiques

Les verrous adressés sont de faire évoluer une architecture logicielle donnée pour la rendre conforme à un style architectural, de garantir une architecture bien formée à l'issue de cette évolution et de spécifier, automatiser et réutiliser l'évolution architecturale.

En effet, il est à noter la quasi-absence de solutions formelles et automatisées pour mener des besoins récurrents de restructuration. Les architectes logicielles s'appuient sur leurs connaissances et celles de leurs équipes pour gérer cette évolution. La situation est dommageable quand ces besoins de restructuration sont rencontrés de manière récurrente. La recherche sur de tels principes pour guider l'évolution architecturale a reçu relativement peu d'attention, malgré les promesses de maîtrise de coûts et autres défis liés au changement [Garlan& 09]. Les restructurations à grande échelle, lorsqu'elles sont récurrentes, gagneraient pourtant à être analysées, formalisées et mises à disposition comme des solutions éprouvées. A titre d'exemple, le besoin de restructurer une architecture monolithique vers une architecture client-serveur est un besoin fortement récurrent. Ces travaux visent à formellement analyser et spécifier de telles restructurations pour assurer une architecture cible cohérente et fiable. Ils visent également à expliciter ces solutions d'évolution afin de les réutiliser à l'infini dans des situations d'évolution identiques.

Les contributions sont présentées ci-après :

5.4. Concepts et opérations pour l'évolution

Pour proposer une solution d'évolution automatique et réutilisable, il est nécessaire d'avoir les éléments clés sous forme d'entités de première classe. Cette phase est nécessaire pour faire une analyse et une évolution automatisée. Comme nous visons l'introduction d'un style architectural sur une architecture par restructuration, nous devons spécifier de manière formelle :

- *Un style architectural:* selon Fielding [Fielding 00], "*an architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style*". En imposant des contraintes sur des familles d'architectures logicielles qu'ils décrivent, ils peuvent contraindre leur évolution. La restructuration est un cas bien adapté. Ces contraintes architecturales permettent l'analyse automatisée d'architectures conformes au style architectural [Garlan& 94].
- *Les dépendances structurelles :* pour aborder correctement la restructuration architecturale, nous devons prendre en compte toutes les dépendances structurelles entre les ports de composants. Ces dépendances peuvent être de deux types. Les *dépendances externes* entre les différents composants sont exprimées à l'aide de *connecteurs*, d'*attachements* et de *bindings*. Les *dépendances internes* entre les ports d'un même composant ne sont généralement pas explicitement matérialisées au niveau architectural. En plus des concepts majeurs des ADLs tels que configuration, composant, connecteur, interface ou encore port (confère Section Chapitre 3.

1.2), nous avons enrichi le métamodèle de l'ADL considéré COSA[12] [Smeda& 05] par une relation de *dépendance interne*. Il s'agit d'une *dépendance structurelle*. Cette relation est exclusivement définie au sein d'un composant. Elle permet de spécifier toutes les dépendances internes entre les ports d'un composant. La dépendance structurelle permet de créer automatiquement une chaîne de connexions si le composant est appelé à être scindé et que des ports reliés se retrouvent dans des composants séparés. A l'inverse, si des composants sont fusionnés, les connecteurs les reliant sont transformés en dépendances structurelles.

- *Les opérations d'évolution :* ont été définies comme des entités de première classe, i.e. instanciables :
  - *Opérations atomiques :* en considérant *e* comme étant un élément architectural, les opérations sont : *Create(e), Delete(e), Move(e)* qui assurent respectivement la création, la suppression de *e* de son parent, et ajout dans sa nouvelle destination.
  - *Opérations composées : Split(), Merge(), Move Out(), Move In(), Delegate()* assurent respectivement la division d'un élément en deux ou plusieurs éléments ; la fusion de deux ou plusieurs éléments en une seule entité ;le déplacement avancé d'un élément afin de le remonter d'un niveau, tout en préservant la cohérence du système ; le déplacement avancé d'un élément afin de le descendre d'un niveau, tout en préservant la cohérence du système ; la création d'un chemin afin de disposer d'un port existant à un autre emplacement du système.

Ces opérations sont génériques. Elles peuvent donc recevoir en paramètre différents éléments architecturaux, mais elles ne peuvent pas s'appliquer pour tous les types de la même manière :

Tableau 4: Distribution des opérations sur les éléments COSA

	Composant	**Connecteur**	**Configuration**	**Port**	**Role**
Delete	√	√	√	√	√
Create	√	√	√	√	√
Move	√	√	√	√	√
Split	√	∗	×	∗	∗
Merge	√	∗	×	∗	∗
Move Out	√	×	×	×	×
Move In	√	×	×	×	×
Delegate	×	×	×	√	×

Légende : × : l'opération n'est pas définie, √ : l'opération est possible,
∗ : l'opération est possible mais non définie formellement.

Certains cas sont triviaux (on ne peut pas déléguer un Composant ou une Configuration), d'autres moins. Par exemple, la combinaison Split ⊗ Configuration. Rappelons qu'une configuration permet d'introduire un sous-système dans un Composant ou un Connecteur (ou de définir la racine du système le cas échéant). Quel que soit le cas, chaque Composant (resp. Connecteur) possèdera au plus une unique Configuration. Il est donc normal de ne pas autoriser volontairement la séparation d'une Configuration. Par une logique identique, la fusion de Configurations est impossible. Notons enfin que le Move In et le Move Out ne s'appliquent pas non plus pour les mêmes raisons : éviter que deux Configurations ne se retrouvent dans un même Composant ou Connecteur, forçant alors l'architecte à revoir la distribution de tous les éléments dans ces deux Configurations.

[12] Cet ADL a été choisi pour l'existence d'une couche d'abstraction méta, permettant de créer de nouveaux concepts instanciables.

Les opérations sont ainsi des entités de première classe instanciables grâce au framework EMF, comme illustré ci-après :

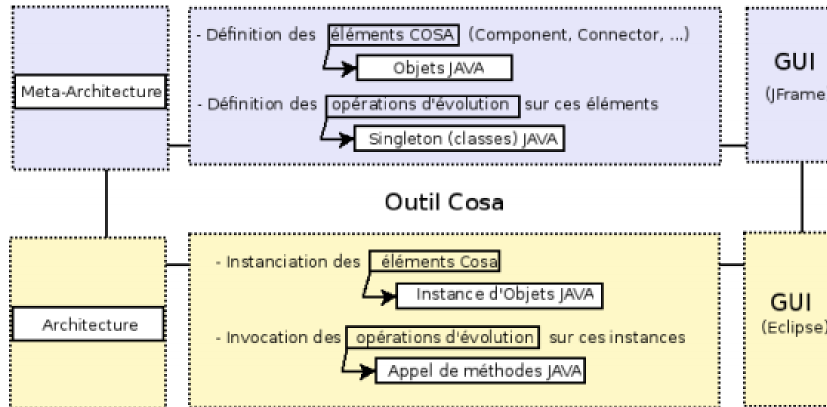


Figure 26: Parallèle entre l'instanciation COSA et Java.

Nous avons fait le choix de représenter chaque opération d'évolution par le design pattern du Singleton en Java. Chaque Singleton possède une unique méthode : *execute()*. Les paramètres de cette méthode permettent de déterminer le contexte ainsi que les éventuels paramètres nécessaires. En définissant chaque opération comme une classe et non comme une méthode, nous permettons de définir un héritage entre opérations afin de créer de nouvelles opérations étendant la mécanique des opérations précédentes. Chaque opération a une logique opérationnelle. Son exécution prend en compte les contraintes du méta-modèle pour les maintenir sur l'élément architectural évolué. Une opération d'évolution ne garantit cependant pas la cohérence globale de l'architecture. Seul le patron d'évolution peut le faire.

5.5. Le patron d'évolution

Nous définissons un patron d'évolution comme une séquence d'opérations d'évolution qui permet de faire évoluer une architecture logicielle d'un état cohérent à un autre état cohérent.

a. Le processus d'évolution semi-formel

Nous illustrons le concept au travers d'une restructuration d'une architecture logicielle pour introduire le style architectural Client-Serveur C&S. Ce patron est principalement une séquence de restructurations localisées menant ensemble à une architecture finale cohérente (formalisé par un diagramme d'activités UML en Figure 27).

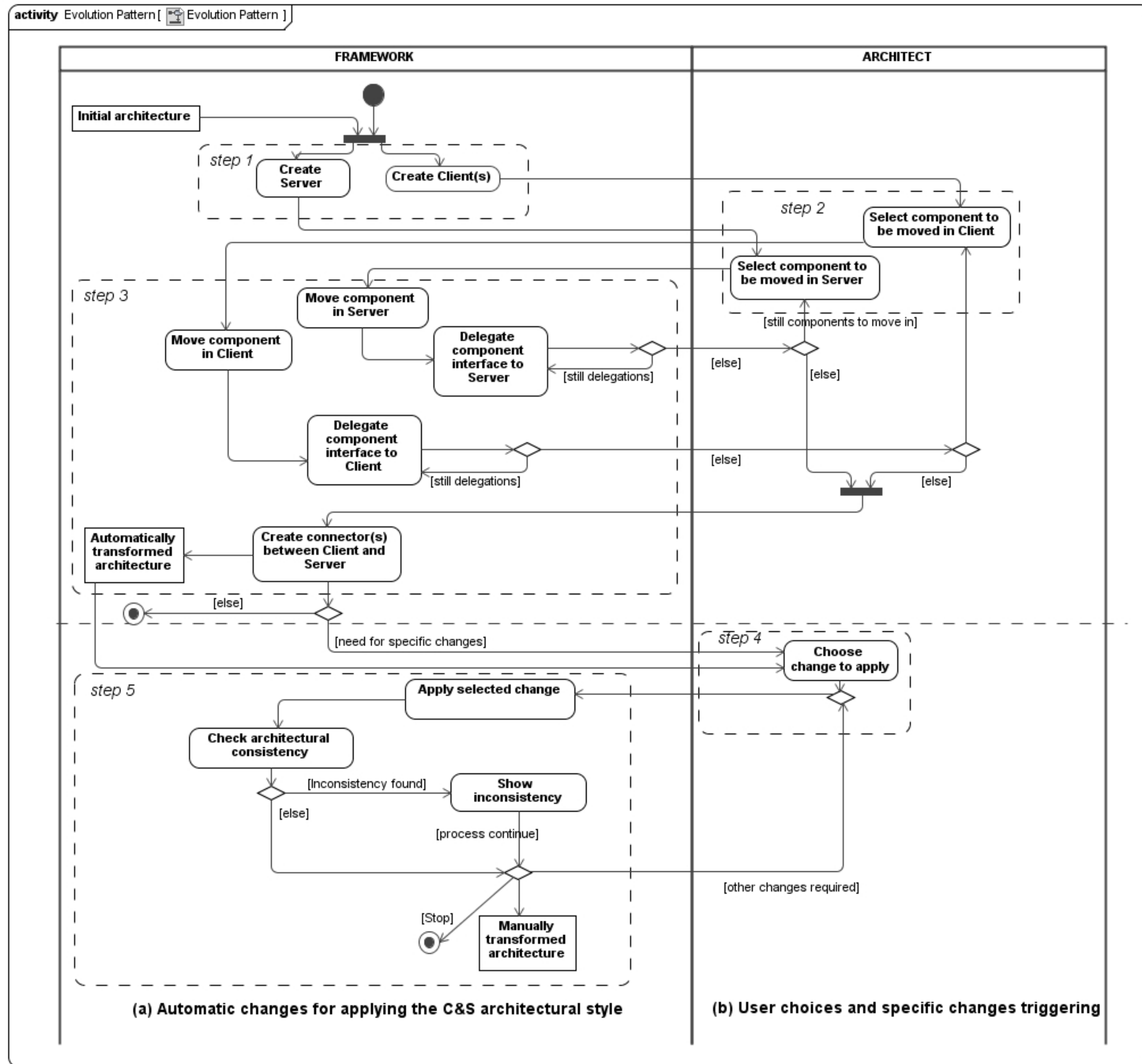


Figure 27 : Le patron d'évolution Client-Serveur : application du style architectural C&S.

Le patron est semi-automatisé. La partie gauche est la partie automatisée. Ce sont toutes les actions qui peuvent être exécutées automatiquement. Lorsqu'une décision doit être prise, l'intervention de l'architecte est indispensable. Nous distinguons 5 phases, chacune entourée d'une ligne en pointillés : les étapes 1 à 3 représentent la première phase obligatoire conforme au style architectural C&S. La phase 2 est facultative et permet à l'utilisateur d'appliquer des modifications spécifiques. La phase 4 revient à l'architecte pour faire des choix architecturaux émanant de la phase 3. La phase 5 finalise les derniers changements. Elle a pour principal objectif de vérifier la cohérence globale et finale de l'architecture à l'issue de la restructuration.

b. Validation par la théorie des graphes

Une condition préalable à l'automatisation du support pour l'application et l'analyse des patrons d'évolution architecturale est la capacité de spécifier avec précision et sans ambiguïté les descriptions d'architecture, les styles architecturaux et les restructurations architecturales. Pour ce faire, nous nous appuyons sur la théorie de la transformation des graphes. L'analogie entre la restructuration de l'architecture logicielle et la transformation des graphes est tout à fait naturelle : une architecture

logicielle appartenant au point de vue C&C peut être exprimée sous la forme d'un graphe contenant un ensemble de composants reliés entre eux par des connecteurs. Les transformations graphiques nous permettent d'exprimer l'évolution de ces graphes architecturaux de manière précise. De plus, il permet d'analyser et de raisonner formellement les évolutions architecturales. Pour spécifier formellement l'ADL, les styles architecturaux et les restructurations architecturales requises pour introduire ces styles, nous avons utilisé AGG[13], un outil de transformation de graphes développé en Java.

La première étape est de formaliser le méta-modèle de l'ADL considéré (exemple du diagramme de composants d'UML 2 en Figure 28). Il est formalisé comme un type de graphe. Toute architecture est alors un graphe conforme à son type de graphe.

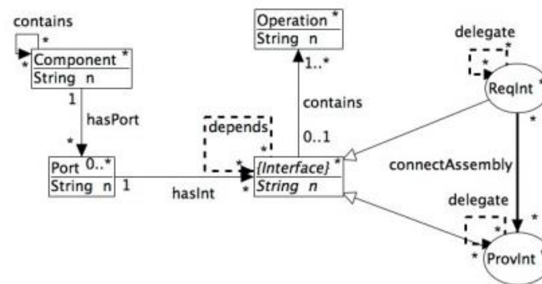


Figure 28 : Graphe type sous AGG du diagramme de composants d'UML 2.

Toutes les contraintes de l'ADL ne pouvant pas être formalisé sur le type de graphe, nous les avons exprimées sous forme d'invariants de graphes couplés au type de graph. Par exemple, un *depends-link* est uniquement autorisé entre deux interfaces appartenant à un port différent du même composant ou encore un *connectAssembly-link* n'est autorisé qu'entre des interfaces d'un type différent (entre une interface requise et une interface fournie) et appartenant à des composants différents qui ne sont pas connectés par un *contains-link*, c'est-à-dire que deux composants ne peuvent être connectés en même temps (via un connecteur) et contenus dans un autre. Ceci s'exprime formellement dans une contrainte de graphe en prenant la négation de la contrainte atomique de la Figure 29 :

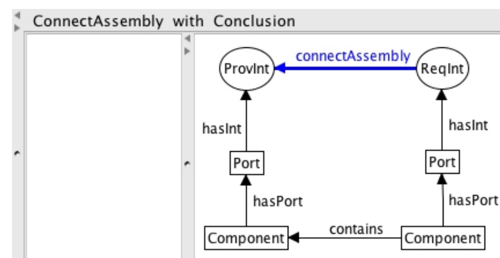


Figure 29 : Contrainte atomique.

Pour formaliser un style architectural, nous procédons exactement de la même manière que pour la formalisation de l'ADL : nous étendons le type de graphe de l'ADL et ajoutons des invariants de graphes supplémentaires qui expriment les contraintes supplémentaires imposées par le style architectural.

[13] http://user.cs.tu-berlin.de/˜gragra/agg/

Enfin, le patron d'évolution est également formalisé, non pas par un type de graphe (celui-ci est adapté à l'expression de contraintes structurelles et statiques) mais par la notion de *transformation de graphes*. Elle prend un graphe en entrée et en produit un autre en sortie. Elle est spécifiée au moyen d'une règle de transformation de graphe qui doit être conforme au type de graphe et à toutes les contraintes du graphe.

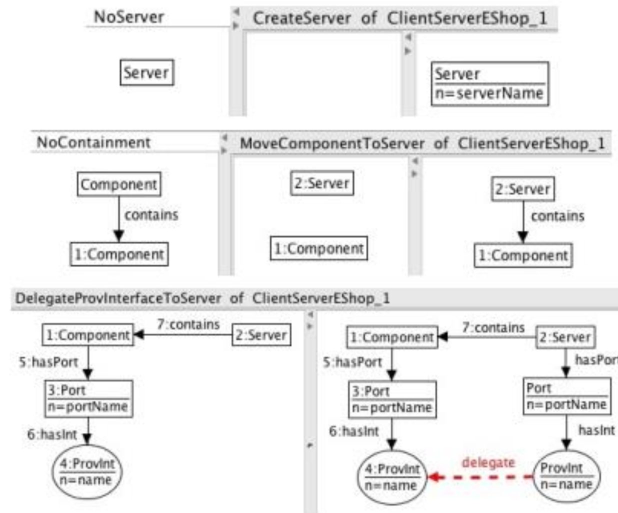


Figure 30 : Trois de règles de transformations pour déplacer un composant au sein d'un serveur.

Le patron d'évolution est ainsi formalisé par une séquence de règles de transformations, à l'image des trois règles en Figure 30.

Une fois le patron exécuté, les résultats sont analysés de manière automatique et formelle : une première analyse triviale consiste à vérifier si une architecture donnée est bien conforme à la spécification de l'ADL, et si l'architecture est conforme à un style architectural. Cette vérification est directement supportée par AGG : il suffit de vérifier qu'un graphe est conforme à son type, ainsi qu'à toutes les contraintes de graphe spécifiées. Une deuxième analyse est le support intégré d'AGG pour l'analyse des paires critiques (CPA) entre les règles de transformation de graphes. Une telle analyse permet de détecter les conflits parallèles et les dépendances séquentielles entre les paires de règles de transformation, guidant ainsi l'architecte dans leur identification et facilitant son intervention. Le résultat de cette analyse automatisée est illustré à la Figure 31 où les flèches en pointillés entre les règles de transformation représentent les dépendances séquentielles et les flèches solides les conflits parallèles :

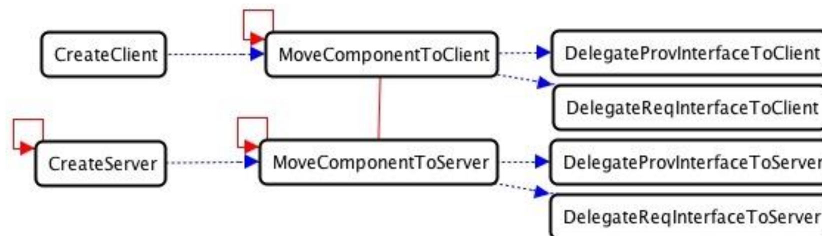


Figure 31 : Analyse par paire critique des règles de transformation.

CreateClient et *CreateServer* sont indépendants l'un de l'autre ; ils peuvent être appliqués en parallèle sans aucun dommage. *CreateServer* est en conflit parallèle avec lui-même car un seul serveur peut

être introduit dans l'architecture client-serveur. Déplacer le composant vers le client et déplacer le composant vers le serveur ont un potentiel conflit parallèle si l'on essaie de déplacer le même composant dans le client et dans le serveur. Déplacer le composant vers le serveur (respectivement le client) est en conflit avec lui-même parce qu'on ne peut pas déplacer le même composant deux fois.

5.6. Contexte des travaux (thèses, M2, contrat, collaborations)

- Collaboration avec Tom Mens lors du semestre passé, durant mon CRCT 2008-2009, en tant que chercheur invité au sein de son équipe.
- **2010** : Aurélien LANSMANE, Master 2 de de l'Université de Mons, Belgique. « Evolution des Architectures Logicielles dans le langage COSA »

- Taux d'encadrement : 50%.

5.7. Brève synthèse des travaux fondateurs

Parmi les ADLs majeurs, citons ACME, Aesop [Garlan& 94], C2 [Medvidovic & 96] ou Rapide [Luckham& 95]. Beaucoup d'autres ADLs ne supportent aucune évolution architecturale (MetaH [Vestal& 93], Unicon [Shaw& 95], Darwin [Magee& 95] ou UML 2 [MOG 07]). Parmi les propriétés d'évolution souhaitables, nous nous intéressons particulièrement à la capacité d'exprimer et d'appliquer, par le biais de l'évolution, des styles architecturaux [Medvidovic& 00].

Ceci dit, différents travaux se concentrent sur des sujets qui font partie de notre cadre d'évolution architecturale. La contribution de nos travaux est double : le concept de *patron d'évolution* et sa spécification formelle ainsi que son analyse par transformation de graphe pour introduire et vérifier les styles architecturaux. Nous avons également implémenté nos résultats dans un outil de modélisation architecturale pour COSA ADL. Le Metayer [Le Metayer 98] a utilisé la théorie de la transformation des graphes pour décrire l'architecture logicielle et le style architectural comme des grammaires de graphes. Il a proposé d'utiliser un coordinateur en termes de règles qui doivent être vérifiées statiquement, par opposition à notre utilisation de modèles d'évolution qui introduisent des styles architecturaux. Wermelinger & al [Wermelinger& 02] ont utilisé la théorie de la transformation des graphes pour présenter une base algébrique pour la reconfiguration d'architectures logicielles. Grunske [Grunske 05] a formalisé les refactorings architecturaux sous forme de règles de transformation hypergraphique qui peuvent être appliquées automatiquement.

L'idée de considérer les styles architecturaux comme des « modèles d'évolution » architecturaux typiques a été introduite par les travaux de thèse de Olivier Le Goaer [Le Goaer 09] qui ont proposé le modèle SEAM qui introduit le concept de modèle d'évolution. Garlan [Garlan 08] défini un style d'évolution comme un ensemble de chemins d'évolution parmi les différents types de systèmes.

McVeigh et al [McVeigh& 06] proposent des opérations de première classe qui expriment et saisissent les changements architecturaux. La principale différence réside dans le type d'opérations d'évolution qui sont fournies et les ADL pour lesquelles elles sont supportées. Une autre différence est que nous utilisons ces opérations d'évolution comme des blocs de construction élémentaires pour créer des modèles d'évolution plus complexes afin d'introduire des styles architecturaux. Dans Noppen et Tamzalit [Noppen& 10] nous avons été au-delà du patron d'évolution en proposant un cadre pour adapter les processus d'évolution en fonction de certains traits architecturaux souhaités en les recherchant dans une base de connaissances architecturale donnée.

5.8. Positionnement et bilan

Nous considérons qu'un modèle d'évolution explicite sur spécifications permettrait une réutilisation future, réduisant ainsi les coûts et les risques de l'évolution architecturale à long terme. Un exemple typique d'un tel schéma récurrent est la restructuration d'une architecture monolithique d'un système existant en une architecture client-serveur distribuée. Nous considérons dans nos travaux la restructuration d'une architecture qui doit respecter les contraintes d'un style architectural (ou plusieurs), tel que le style architectural *Client-Serveur*. La première contribution de nos travaux est de rendre une architecture conforme à un style architectural par l'application de *restructurations architecturales*. Nous proposons ainsi d'utiliser les styles architecturaux comme un moyen discipliné pour guider la restructuration d'architectures. Ces dernières sont analysées formellement afin d'assurer leur complétude et leur cohérence. Pour cela, nous nous appuyons sur la théorie de la transformation de graphes pour offrir une analyse formelle de telles évolutions. La deuxième contribution consiste à pouvoir réutiliser ces restructurations en les explicitant et les formalisant. Nous avons ainsi proposé des opérations élémentaires d'évolution réutilisables avec la possibilité d'en composer de nouvelles, réutilisables également.

A notre connaissance, aucun ADL ne se propose des supports intrinsèques pour guider l'évolution par l'application de styles architecturaux sur une architecture donnée. Les mécanismes d'évolution proposés par les ADL sont généralement liés à des outils support, et donc difficilement réutilisables lorsqu'une situation d'évolution architecturale similaire se produit.

Ma contribution : a porté sur la connaissance des ADLs et des travaux d'évolution dédiés, toutes les spécifications semi-formelles des architectures logicielles, des styles architecturaux, le méta-modèle et le processus de migration.

Validation : nous avons validé nos contributions sur l'ADL UML 2.0. Ce dernier étant ambigu et informel dans les spécifications qu'il offre, nous avons en premier restreint les spécifications. Nous avons explicité, analysé et validé le processus de restructuration d'une architecture monolithique UML 2.0 vers une architecture client-serveur grâce à l'outil de transformation de graphes AGG[14]. Nous avons également validé ce travail en fournissant une validation pratique en implémentant toutes les opérations d'évolution et le processus d'évolution au niveau méta et appliqué à des architectures exemples. L'outil utilisé est COSA-Builder, un plug-in Eclipse pour le langage de description architectural COSA [Smeda& 05], dans le cadre du stage de Master 2 d'Aurélien Lansmanne de l'Université de Mons (Belgique).

Valorisation : ce travail a donné lieu à trois publications, dont une conférence de rang B, IEEE ECBS 2010 (référence [42] du CV détaillé).

6. Conclusion

Ces travaux, menés avec mes collaborateurs, sur les architectures logicielles à base de composants ne se sont pas poursuivis. Cela est dû en grande partie à l'avènement de l'approche orientée service qui a pris le pas.

[14] http://user.cs.tu-berlin.de/~gragra/agg/

Chapitre 4.

Architectures logicielles à services (SOA, SaaS mutualisé, micro-services) : contributions

SOMMAIRE

1. Introduction ___ 76
1.1. Architectures Logicielles et notion de Service ___ 76
1.2. Restructuration, externalisation, migration autour de la notion de *service* ___ 84
2. Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA ___ 85
2.1. Fiche d'identité ___ 85
2.2. Synthèse ___ 85
2.3. Verrous scientifiques ___ 87
2.4. Interconnexion de WF avec les Patrons de coopération à base de services ___ 87
2.5. Support de la flexibilité des modèles de WFIO ___ 92
2.6. Contexte des travaux (thèses, M2, contrat, collaborations) ___ 93
2.7. Brève synthèse des travaux fondateurs ___ 93
2.8. Positionnement et bilan ___ 94
3. Externalisation du modèle SaaS mutualisé et de sa gestion ___ 96
3.1. Fiche d'identité ___ 96
3.2. Synthèse ___ 96
3.3. Verrous scientifiques ___ 97
3.4. Gestion de la Variabilité des besoins de locataires ___ 97
3.5. Modélisation de la variabilité : nos contributions ___ 100
3.6. Contexte des travaux (thèses, M2, contrat, collaborations) ___ 104
3.7. Brève synthèse des travaux fondateurs ___ 104
3.8. Positionnement et bilan ___ 105
4. Capitalisation de migrations vers des architectures micro-services ___ 106
4.1. Fiche d'identité ___ 106

4.2. Synthèse ___ 106
4.3. Verrous scientifiques ___ 107
4.4. Migration de monolithes vers les micro-services : recommandations techniques ___ 108
4.5. Migration de monolithes vers les micro-services : recommandations fonctionnelles ___ 110
4.6. Contexte des travaux (thèses, M2, contrat, collaborations) ___ 112
4.7. Brève synthèse des travaux fondateurs ___ 112
4.8. Positionnement et bilan ___ 112

5. Conclusion ___ 113

1. Introduction

Après une phase de travaux de recherche académiques autour des architectures logicielles à base de composants durant 8 ans, j'ai démarré, en 2010, des travaux de recherche sur les architectures logicielles à base de services, suivant ainsi le changement de paradigme qui s'était opéré des composants vers les services. Ce chapitre regroupe une sélection de travaux que j'ai menés avec différents collaborateurs. Ces travaux sont liés à deux thèses que j'ai eu le plaisir de co-encadrer, l'une académique et menée par Souad Boukhedouma autour de l'adaptabilité des workflows d'entreprises avec une approche SOA (Chapitre 4. 2.), l'autre industrielle et menée par Ali Ghaddar autour des problématiques de gestion du SaaS mutualisé (Chapitre 4. 3.). A ces travaux s'ajoutent ceux que je mène en collaboration avec Jean-Philippe Gouigoux, Directeur Technique de l'éditeur logiciel MGDIS, autour de la migration d'applications monolithes vers une architecture micro-services (Chapitre 4. 4. . *Ces travaux ont démarré en 2010 avec une continuité à ce jour pour une partie d'entre eux.*

Avant de présenter les travaux, je pose les notions et concepts nécessaire à la présentation des travaux :

1.1. Architectures Logicielles et notion de Service

Mon objectif n'est pas de faire une présentation exhaustive du et autour du service. Je souhaite poser les concepts et notions utilisés dans les différents travaux menés :

a. Notion de service

La notion de *service*, parue dans les années 2000 [Pilioura& 01], a répondu aux limitations du *composant*. Les composants sont certes des entités préconçues, réutilisables et composables mais, sous réserve de leur compatibilité d'assemblage. Les services ne se connaissent pas a priori mais la coopération a été prévue dans un cadre contractuel leur permettant de se découvrir en temps voulu. Fondamentalement, un service *est* un composant. Cependant, une architecture basée sur les services propose un cadre d'assemblage moins figé et offre un cadre plus dynamique qu'une architecture basée sur les composants. La notion de service a également pris de l'ampleur avec la popularisation des systèmes distribués, notamment au sein des entreprises.

Le service représente une entité logicielle fonctionnelle déployée et invocable à distance [Hock-Koon 11]. Un service est lié à un fournisseur de services et est utilisé par des clients à partir de sa description. Celle-ci décrit les propriétés du service qui représentent sa particularité métier et permet aux clients de décider si ce service correspond à leurs besoins. De nouvelles applications peuvent ensuite être assemblées en construisant de nouveaux services et récursivement composer les services

existants pour atteindre la fonctionnalité désirée. Une composition de services est ainsi fournie à d'autres applications en tant que service. Les services sont caractérisés par une interface bien définie décrivant les propriétés fonctionnelles et non-fonctionnelles du service [Mietzner 10]. Un service peut effectuer des actions allant de simples requêtes à des processus métiers complexes. Les services permettent d'intégrer des systèmes d'information hétérogènes en utilisant des protocoles et des formats de données standardisés, autorisant ainsi un faible couplage et une grande souplesse vis-à-vis des choix technologiques effectués [Papazoglou 03].

b. Une implémentation de service : les services Web

Le service en tant qu'entité logicielle, en plus de leur interface, ont une implémentation. Les *services Web* sont certainement la technologie la plus connue pour l'implémentation de services. La pile technologique de services Web (Web Service stack) [Curbera& 05] fournit un ensemble de technologies et de spécifications. Ainsi, les services sont appelés services Web dans cette pile technologique. Un certain nombre de langages et d'outils sont définis, permettant un large usage des services Web : le langage de description de services Web WSDL (Web Service Description Langage) [Chinnici& 07] généralement écrit en XML ; le standard Web Service Policy (WS-Policy) [Bajaj& 06] pour décrire les propriétés non fonctionnelles des services ; le registre de services UDDI (Universal Description Discovery and Integration) [Curbera& 05] ou encore les protocoles standards de communication tels que SOAP (Simple Object Access Protocol) [Curbera& 05], lui-même basé sur s'appuie sur des standards de communication comme HTTP, SMTP et FTP.

En plus d'un panel de standards pour les spécifier, les services Web peuvent être composés par *orchestration*, qui correspond à un schéma de collaboration centralisée, ou par *chorégraphie*, qui correspond à un schéma de collaboration distribuée [Peltz 03] [Pahl& 06] (confère Figure 32).

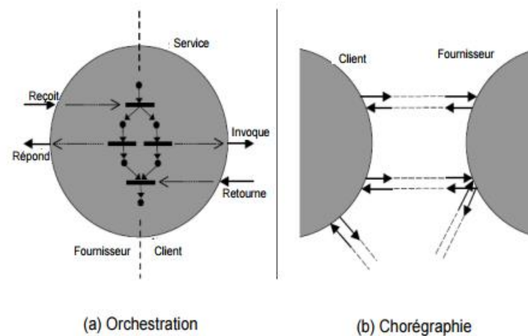


Figure 32 : Orchestration et chorégraphie de services [Pahl& 06].

Cette composition de services peut être *statique*, donc prédéfinie et effectuée lors de la conception, ou *dynamique*. Cette dernière permet une réactivité importante sans arrêter le système, contrairement à la première.

c. Architecture orientée service

Le terme Service-Oriented Architecture (SOA), ou architecture orientée service (AOS), est apparu début 2000 [Papazoglou 03] [Papazoglou& 07]. L'architecture orientée service a principalement émergé comme un style architectural pour modéliser et construire des systèmes informatiques distribués, essentiellement comme recommandations de conception.

Elle a connu une large adoption au regard de son concept pivot de *service*. Une caractéristique importante d'une SOA est que son modèle d'architecture ne se base sur aucune technologie d'implémentation spécifique. Les services peuvent être distribués par exemple sur des machines différentes, dans des entreprises et des pays différents [Curbera& 05] [Papazoglou 03]. Elle propose également une organisation en couches, permettant une bonne séparation des préoccupations :

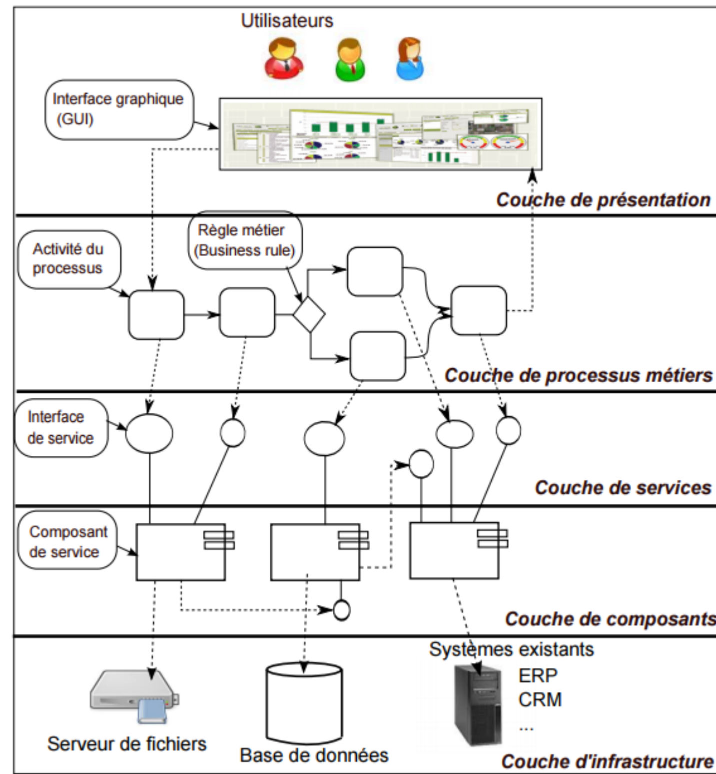


Figure 33 : Les couches d'une SOA [Arsanjani& 07].

Les services d'une SOA de la couche de services (Figure 33, qui se rapproche de la Figure 4) sont conçus comme des modules autonomes qui peuvent être facilement réutilisés grâce à une organisation dédiée (Figure 34) [Bellwood& 02] [Curbera& 02]. Dans une SOA, l'invocation d'un service peut être statique (au moment de la conception ou du déploiement de l'application) : le client interroge le registre pour sélectionner les services souhaités. L'invocation peut également être dynamique à travers l'utilisation des mécanismes spécifiques [Charfi& 07] [Ardagna& 07] [Koning& 09].

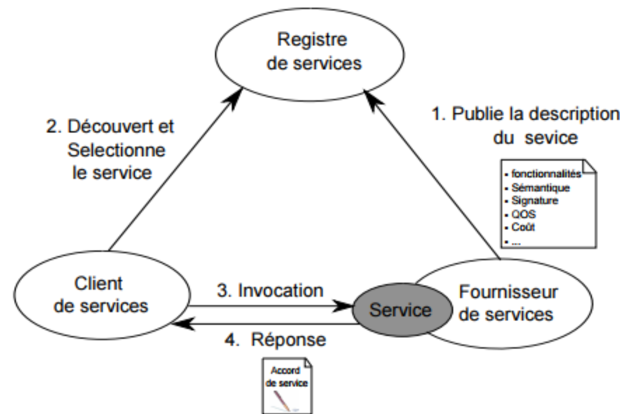


Figure 34 : Organisation de l'architecture orientée services.

Pour la mise en œuvre d'une SOA, différentes solutions techniques sont envisageables. Les services Web sont les candidats idéaux notamment pour la mise en œuvre une architecture orientée services sur des infrastructures hétérogènes et distribuées [Newcomer 05]. Il existe plusieurs plateformes de développement pour implémenter les services et les déployer sur Internet en utilisant différents langages de programmation tels que Java, C#, C, PHP...

d. Le Cloud computing

Selon Feuerlicht, le Cloud a redéfini les bases sur lesquelles l'industrie informatique a fonctionné pendant des décennies [Feuerlicht 10]. Il est devenu incontournable [Geelan 09] [Vaquero& 08] [Tograph& 08] [Buyya& 09][Mell& 11]. Il s'appuie en grande partie sur les résultats de recherches effectués sur les technologies de virtualisation [Barham& 03], l'informatique distribuée et utilitaire [Ross& 04], les grilles de calculs [Boag& 03] et par la suite, sur le web et l'architecture orientée service [Papazoglou 03] [Papazoglou& 07]. La Figure 35 montre un aperçu général des aspects du Cloud [Du& 10] [Khajeh-Hosseini 10], notamment ses caractéristiques essentielles, facilitateurs principaux, modèles de fournitures et modes de déploiement.

(a) Les caractéristiques essentielles	Mutualisation			
	Elasticité		Self-Service	
	Accès simple via le réseau		Facturation à l'usage	
(b) Les facilitateurs pricipaux	Grilles de calculs et informatique utilitaire			
	Virtualisation		Client riche	
	Architecture orientée service		Couverture large du réseau	
(c) Les modèles de fournitures	Un Logiciel sous forme d'un service (SaaS)			
	Une Plateforme sous forme d'un service (PaaS)			
	Une Infrastructure sous forme d'un service (IaaS)			
(d) Les modes de déploiement	Privé	Public	Hybride	Virtuellement privé

Figure 35 : Aperçu général du Cloud.

Je ferais le focus uniquement sur les aspects du Cloud abordés dans mes travaux avec mes collaborateurs, notamment ceux dédiés au modèle SaaS mutualisé et les micro-services :

(a) Parmi les caractéristiques essentielles, nous nous sommes intéressés à :

   a. La *mutualisation :* les ressources du Cloud sont mises en commun et mutualisées afin de servir de multiples clients (utilisateurs finaux, organisations, éditeurs de logiciels, etc.). Cette mutualisation peut intervenir à des niveaux multiples qu'il s'agisse des services infrastructurels ou applicatifs. Dans le dernier cas, qui s'applique généralement au Cloud applicatif représenté par le modèle *SaaS*, on parle souvent d'une architecture mutualisée [Chong& 06] (*multi-tenant architecture* en anglais). Grâce à cette mise en commun des ressources, ces dernières sont réallouées de façon dynamique en fonction de la demande et les accords du niveau de service (*Service Level Agreement SLA*) [Pervez& 10]. Chaque client est ainsi assuré d'atteindre ses objectifs de performances définis dans le cadre de son contrat.

   b. L'*élasticité* : dans le Cloud, de nouvelles ressources peuvent être rapidement mises à disposition des clients pour supporter l'augmentation de leur charge de travail. De même, ces ressources peuvent être rapidement libérées lorsqu'elles ne sont plus nécessaires.

   c. Le *self-service* : dans le Cloud, un client peut unilatéralement avoir accès de manière automatisée à des ressources informatiques (serveur, stockage, réseau, application, etc.), et en disposer tant qu'il en a besoin sans avoir à passer par de longues et complexes étapes de configuration ou d'intervention manuelle de la part du fournisseur. Dans les modèles de fourniture SaaS et PaaS, cette caractéristique du Cloud est la plus mature et où l'entreprise cliente s'affranchit d'un grand nombre de contraintes pour la mise en place de ses applications.

(b) Parmi les *facilitateurs principaux*, nous nous sommes intéressés à l'*architecture orientée service* précédemment présentée.

(c) Parmi les *modèles de* fournitures, nous nous sommes intéressés au modèle *SaaS*. Les services du Cloud, très nombreux, sont généralement divisés en trois modèles de fournitures [Armbrust& 10][Mell& 11][Subashini& 11] dont chacun représente une couche d'abstraction qui s'ajoute à la pile technologique nécessaire pour mettre en place une application (confère Figure 36) : SaaS (*Software as a Service*), PaaS (*Platform as a Service*) et IaaS (*Infrastructure as a Service*). Il est également nommé le *modèle SPI du Cloud*, S pour SaaS, P pour PaaS et I pour IaaS [Mell& 11].

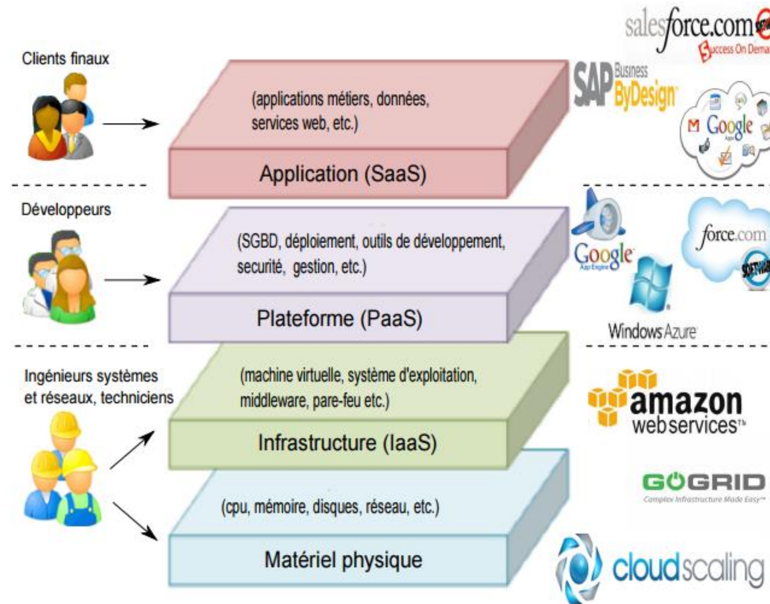


Figure 36 : Les modèles de fourniture du Cloud ou le modèle SPI.

Ces couches de l'architecture du Cloud [Zhang& 10] ne représentent pas juste une encapsulation des ressources accessibles à la demande, mais elles définissent également un nouveau modèle de développement d'application. Chaque couche établie un rapport de contrôle différent sur la pile technologique entre le client et le fournisseur.

e. Le SaaS (Software-as-a-Service) et ses niveaux de maturité

Le modèle SaaS est un type des services du Cloud qui permet de mettre à disposition un logiciel via un navigateur Web. Il représente le modèle le plus mature du Cloud. Le logiciel est donc utilisé par Internet et ne nécessite plus d'être installé en local. L'environnement du Cloud (en termes d'infrastructures et de plateformes) décharge les utilisateurs et leur assure l'exécution et la maintenance de l'ensemble des ressources. Les travaux dédiés identifient quatre politiques de gestion de multiples clients, chacune appartenant à un niveau de maturité du SaaS [Chong& 06] [Kwok& 08] [Liu& 10] (Figure 37). Les attributs clés différenciant ces niveaux de maturité sont [Kang& 10] :

- *La configuration :* l'adaptation aux besoins spécifiques des utilisateurs se fait avec des choix de configurations et non par modification du code.
- *La mutualisation :* une instance unique de l'application est utilisée pour servir de multiples utilisateurs en même temps.
- *Le passage à l'échelle :* capacité à délivrer ces fonctionnalités avec un temps de réponse constant et des coûts opérationnels « acceptables », même si le nombre d'utilisateurs ou de données augmente de manière importante.

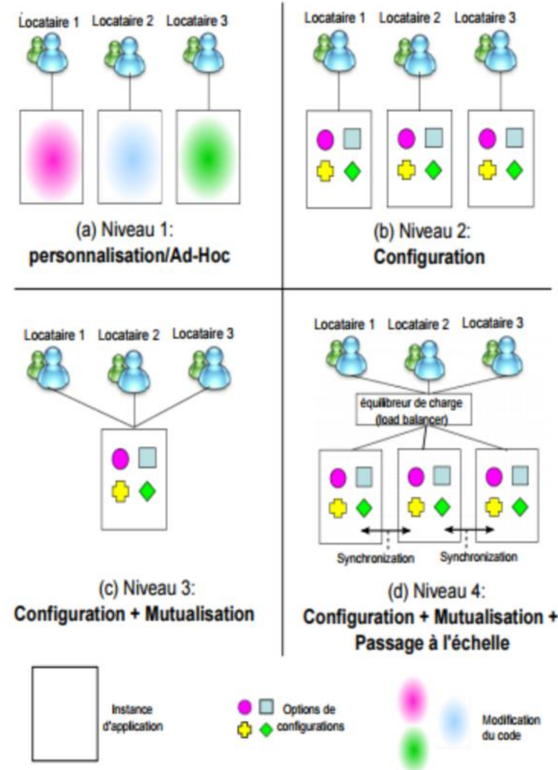


Figure 37 : Les niveaux de maturité du SaaS.

Les quatre niveaux de maturité du SaaS sont :

- *Niveau 1 – Personnalisation/Ad-Hoc :* chaque client a sa propre version personnalisée de l'application déployée séparément sur un serveur (physique ou virtuel) dédié (Figure 37 (a)). Ce niveau de maturité de SaaS correspond dans la majorité de ces caractéristiques au modèle ASP [Kwok& 08]
- *Niveau 2 – Configuration :* il ajoute au Niveau 1 de la flexibilité au logiciel (Figure 37(b)). Chaque utilisateur aura sa propre instance adaptée par configuration des options fournies par la même version du logiciel. Chaque instance est configurée avant l'exécution. Elle se comporte donc de manière identique pour tous les utilisateurs.
- *Niveau 3 – Configuration et mutualisation :* il ajoute la mutualisation au Niveau 2 (Figure 37 (c)). Tous les utilisateurs (dénommés *locataires* ou *tenant*) exécutent la même instance du logiciel. Leur configuration sont isolées et appliquées dynamiquement, à l'exécution du logiciel, car les identités des locataires ne sont pas connues à l'avance.
- *Niveau 4 – Configuration, mutualisation et passage à l'échelle :* il ajoute la possibilité de passer à l'échelle par rapport au Niveau 3. Il propose de déployer plusieurs instances de l'application mutualisée sur des serveurs à puissances identiques [Armbrust& 10] (Figure 37 (d)). Le passage à l'échelle se fait automatiquement via un équilibreur de charge (*Load Balancer*).

f. Micro-services

Le micro-service est un des nombreux styles architecturaux des architectures logicielles [Newman 15] et un des plus récents [Fowler& 14]. Il peut être vu comme une petite application qui n'a qu'une seule responsabilité, déployée indépendamment, mise à l'échelle indépendamment et testée indépendamment. En soi rien de bien nouveau sur le papier en termes de caractéristiques mais

qui a connu le succès, basé sur une conjonction de faits survenus ensemble dans une période de temps proche : les leçons tirées des échecs de la SOA dans le monde de l'entreprise, la grande démocratisation des systèmes distribués et notamment l'avènement du modèle SPI du Cloud tel que défini par le NIST[15] [Mell& 11] ainsi que l'apparition de la virtualisation [Barham& 03], la contenarisation [Dua& 14], notamment avec Docker en 2003, et le style architectural REST (*Representational State Transfer*) [Fielding 00]. Ce style architectural, relativement récent, permet de concevoir des applications complexes qui sont décomposées en plusieurs processus indépendants et faiblement couplés, souvent spécialisés dans une seule tâche. Les processus indépendants communiquent les uns avec les autres en utilisant des API indépendantes du langage de programmation. L'on lit et entend souvent : *"Micro-services are SOA made right."*

Voyons un peu plus en détails : "*Micro-services are small, autonomous services that work together*" telle est la définition donné par Newman, la référence la plus cité sur les micro-services [Newman 15], même si la parenté revient à Martin Fowler [Fowler& 14]. Un micro-service est :

- *Petit* : mais surtout conçu pour bien faire une et une seule fonction. Quand un code logiciel s'est développé par ajout de nouvelles fonctionnalités, il est parfois difficile au fil du temps de savoir où apporter un changement, généralement en raison d'un code important. Ce fait est amoindri lorsqu'il est modulaire. La cohésion et le principe de responsabilité unique, des notions qui existent depuis longtemps en génie logiciel, sont des notions importantes lorsque l'on pense aux micro-services.
- *Autonome* : un micro-service est une entité distincte. Il doit être faiblement couplé à d'autres micro-services. Il doit également être capable d'être déployé en tant que service isolé sur une Platform-as-a-Service (PaaS) par exemple, et de pouvoir s'y exécuter. Toutes les communications entre les services eux-mêmes se font par des appels réseau, pour renforcer la séparation entre les services et éviter les dangers des forts couplages. Il est pour cela primordial de bien définir les frontières du micro-service et de s'y tenir, notamment lors de changements. Ces frontières sont généralement celles de la fonction métier implémentée par le micro-service.

Une application à base de micro-services est un système constitué de petites unités indépendantes qui collaborent entre elles, ayant chacune un cycle de vie et une responsabilité propre, facilitant leur évolution. Le passage à l'échelle se fait grâce à l'allocation de ressources souhaitées à chaque type de service. Un service avec une forte charge possède plus d'instances d'exécutions qu'un service faiblement sollicité.

Les principaux avantages de l'adoption des micro-services est l'*hétérogénéité des technologies*, car elles sont embarquées à l'intérieur des micro-services, la *résilience*, car un micro-service est (doit être) proprement cloisonné par ses frontières, le *passage à l'échelle*, car les micro-services peuvent être déployés et s'exécuter sur différentes machines distantes, la *facilité de déploiement* et leur *composabilité*.

Une fois les différents concepts et notions nécessaires posés, je présente brièvement l'essentiel des verrous traités par nos travaux, notamment pour des objectifs de restructuration, d'externalisation et de migration :

[15] NIST: National Institute of Standards and Technology https://www.nist.gov

1.2. Restructuration, externalisation, migration autour de la notion de *service*

L'approche SOA, à l'origine de l'avènement de la notion de *service*, a connu beaucoup de revers. Il lui est ainsi reproché de dire trop de choses différentes. Elle a plutôt connu trop d'applications différentes faites de manière souvent liées à des adaptations contextuelles, à des confusions et à des raccourcis vite pris. Parmi les principales raisons d'échec, selon Martin Fowler ainsi qu'au vu de diverses expériences, je présente celles qui me paraissent les plus critiques. Une des premières raisons d'échec auprès des entreprises est l'usage fait des ESB (Enterprise Service Bus) pour intégrer des applications monolithiques. Généralement, la complexité et les fortes dépendances ne sont pas résolues mais déplacées de manière cachée au sein de l'ESB. Une autre raison est la croyance qu'avoir des web services fait que forcément il y a une SOA. Et encore d'autres raisons : l'absence de consensus sur la bonne manière de faire une SOA, l'absence de directives sur la granularité des services ou des directives erronées sur les endroits où séparer un système. En particulier, il y a eu beaucoup de mises en œuvre bâclées de l'orientation service, beaucoup d'échecs qui ont coûté des millions et n'ont apporté aucune valeur ainsi qu'aux modèles de gouvernance centralisée qui empêchent activement le changement. Finalement, peut-être que l'approche SOA, victime de son succès au début, a-t-elle été dévoyée pour des profits économiques, l'éloignant de son objectif initial ?

Dans le cadre des travaux de recherche que j'ai mené avec mes collaborateurs, nous avons été particulièrement attentif à la définition des services, que ce soit pour les travaux dédiés aux workflows d'entreprise avec une approche SOA, pour la gestion du modèle SaaS mutualisé que pour la migration de monolithes vers des architectures micro-services. Je considère ainsi la définition suivante d'un service, en bonne partie inspirée du OASIS Reference Model for SOA[16] : *un service est la mise en œuvre d'une fonctionnalité bien définie, disposant de stratégies pour contrôler son utilisation, fournir un accès via une interface prescrite, fonctionnant indépendamment de l'état de tout autre service et en accord avec les contraintes et les stratégies qui y sont spécifiées*. Je souligne la *fonctionnalité bien définie*, qui est à mon sens, le point essentiel. L'identification et la définition de services partageables et réutilisables est une tâche difficile. Elle reste pour l'instant plus une question d'expériences, voire d'art, qu'une science. Techniquement, n'importe quelle fonctionnalité peut être un service, mais un service doit représenter un concept fonctionnel tangible. La technologie ne devrait pas diriger cette identification.

Définir un service comme une fonctionnalité bien définie avec une responsabilité unique, a été important pour nos travaux de restructurations de workflow de processus d'entreprises avec une approche SOA (Section 2.), d'externalisation sous forme de services la gestion du modèle SaaS mutualisé (Section 3.) et l'analyse de systèmes monolithiques pour en extraire les bons micro-services et assurer une migration réussie (Section 4.). Ces travaux sont présentés ci-après.

[16] https://www.oasis-open.org/committees/soa-rm/

2. Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA

2.1. Fiche d'identité

Titre :	Adaptation et restructurations de Workflow de processus d'entreprises avec une approche SOA
Paradigme(s) :	Business process – Architectures orientées services.
Principales contributions :	. La définition de nouveaux schémas de coopération équivalents à ceux proposés dans la littérature avec un degré de flexibilité supplémentaire, visant à supporter aisément les changements sur les modèles de WFIO. Une démarche de restructuration et d'interconnexion de modèles de WF basée sur le paradigme SOA (Architecture Orientée Service) et un nouveau concept appelé Patron de Coopération à Base de Services (PCBS) permettant de caractériser un modèle de processus WFIO à base de services . La prise en charge de différents aspects de flexibilité de ces modèles formalisée à l'aide d'opérateurs dédiés et un ensemble de patrons d'adaptation. Pour les aspects d'évolution et de réutilisation de modèles, les concepts de patron de coopération généralisé et patron de coopération composite décrivent un aspect de réutilisation des modèles de WFIO pour la construction de nouveaux modèles de WFIO plus complexes.
Mots-clés :	Coopération B2B, Workflow Inter-organisationnel (WFIO), SOA, Service, Restructuration, Interconnexion, Patron de coopération à base de services (PCBS), Patron d'adaptation, Patron de coopération généralisé, Patron de coopération composite, Flexibilité, Adaptation, Evolution, Réutilisation.
Collaborateur(s) :	S. Boukhedouma (doctorante), M. Oussalah (Université de Nantes), Z. Alimazighi (USTHB, Alger)
Mon implication :	Encadrement à 20% de la thèse de S. Boukhedouma.
Cadre de collaboration :	Projet franco-algérien PAI CMEP Tassili
Période :	2010-2015.
Principales publications :	2 revues internationales : IJBIS 2015 et IJITCS 2014 ; 1 chapitre d'ouvrage dans "Uncovering Essential Software Artifacts through Business Process Archeology" 2013 ; 2 conférences de rang B : IEEE RICS 2013 et BIS 2012 ; 3 autres conférences internationales : AICCSA 2013, ICEIS 2012, ICISTM 2012 et 2 conférences nationales : INFORSID 2012 et INFORSID 2011.

Figure 38 : Fiche synthétique.

2.2. Synthèse

Ces travaux s'inscrivent dans le domaine de la coopération B2B (business to business). Le B2B désigne l'ensemble des activités commerciales existantes entre deux entreprises. De manière générale, le B2B concerne tous les moyens utilisés pour mettre en relation ces entreprises et faciliter les échanges de produits, de services et d'informations entre elles. Ce type de coopération a été initialement supporté par les outils de *workflow inter-organisationnels (WFIO)*, des schémas génériques de coopération (appelés aussi architectures de WFIO) ont été clairement identifiés dans la littérature [Van der Aalst 99] [Van der Aalst 00] et reconnus comme des modèles de coopération

assez répandus dans le domaine du B2B. Il s'agit du « *Partage de charge* », l'« *Exécution chainée* », la « *Sous-traitance* », le « *Transfert de cas* », le « *Transfert de cas étendu* » et le WF « *Faiblement couplé* ». Ces schémas de coopération définissent chacun une architecture de WFIO caractérisée par trois dimensions principales : le *partitionnement du processus*, le *contrôle d'exécution* (centralisé, décentralisé, hiérarchisé ou mixte) et *la structure d'interaction* (synchrone ou asynchrone) entre les processus impliqués dans la coopération.

Nos travaux s'intéressent à la coopération planifiée supportée par les concepts et les outils de *WorkFlow Inter-Organisationnel (WFIO)* en nous basons sur les schémas de coopération précédemment cités. Ceux-ci définissent différents modes de partitionnement, de contrôle d'exécution et d'interaction pour l'interconnexion de processus WF issus de différentes organisations. Nous nous sommes focalisés sur deux objectifs étroitement liés :

1. Proposer une approche de restructuration et d'interconnexion de processus métiers pour obtenir des modèles de WFIO suffisamment flexibles (partie gauche de la Figure 39). Il s'agit de la définition de nouveaux schémas de coopération équivalents à ceux proposés dans la littérature avec un degré de flexibilité supplémentaire, visant à supporter aisément les changements sur les modèles de WFIO. Nous avons ainsi proposé une *démarche de restructuration et d'interconnexion de modèles de WF* basée sur le paradigme SOA ; nous avons introduit un nouveau concept appelé *Patron de Coopération à Base de Services (PCBS)* permettant de caractériser un modèle de processus WFIO à base de services selon trois dimensions principales. Aussi, nous avons procédé à une conceptualisation des différents patrons de coopération proposés et une formalisation de chacun d'eux en définissant un opérateur de coopération approprié.
2. Proposer des mécanismes support de la flexibilité de ces nouveaux modèles (partie droite de la Figure 39). Nous percevons ces mécanismes sous trois aspects complémentaires : *adaptabilité*, *évolutivité* et *réutilisabilité*. A cet effet, nous avons proposé et formalisé à l'aide d'opérateurs dédiés, un ensemble de patrons d'adaptation. Pour les aspects d'évolution et de réutilisation de modèles, nous avons introduit les concepts de patron de coopération généralisé et patron de coopération composite qui décrivent un aspect de réutilisation des modèles de WFIO, pour la construction de nouveaux modèles de WFIO plus complexes.

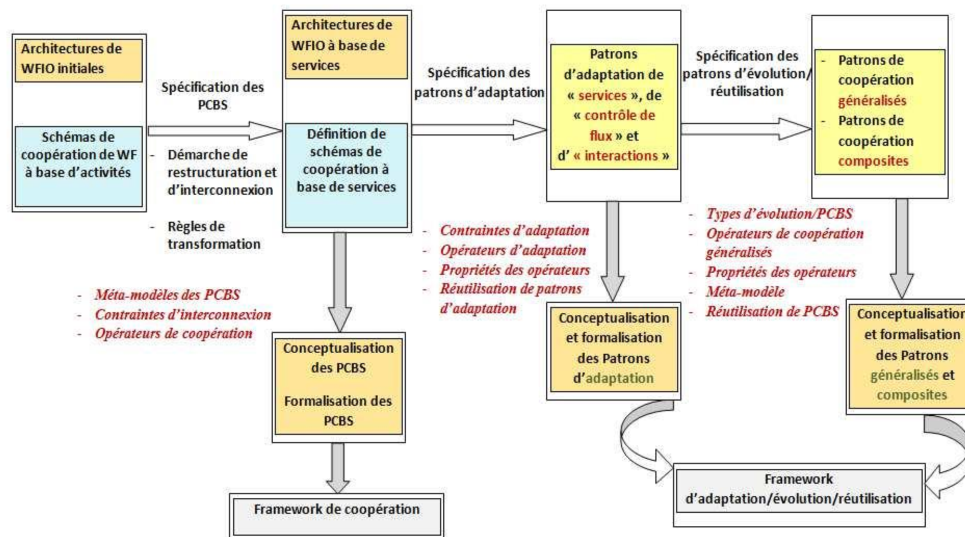


Figure 39 : Schéma global de notre approche.

2.3. Verrous scientifiques

Dans le cadre d'échanges B2B (Business-to-Business), les coopérations planifiées sont supportées notamment par des *WorkFlow Inter-Organisationnel (WFIO)*.

- Le premier verrou abordé est le manque de flexibilité des schémas de coopération des modèles existants de WFIO. Nous le traitons en définissant de nouveaux schémas de coopération plus flexibles en se basant sur le paradigme SOA (Architecture Orientée Service).
- Le second verrou, en lien avec les solutions proposées pour le premier verrou, concerne le manque de solutions pour assurer l'évolution et la réutilisation de modèles de WFIO existants ainsi que leur combinaison pour construire des modèles plus complexes. Nous abordons ce verrou selon l'*adaptabilité*, l'*évolutivité* et la *réutilisabilité* des modèles de WFIO.

Les contributions sont présentées ci-après :

2.4. Interconnexion de WF avec les Patrons de coopération à base de services

Le concept de service est utilisé aussi bien dans les entreprises de services traditionnelles, que dans les entreprises manufacturières et les prestataires publics de services. On parle souvent de l'économie de service dans laquelle les entreprises offrent leurs produits sous forme de services. Particulièrement, les services Web ont émergé comme l'instanciation la plus importante du modèle SOA dans le domaine industriel [Gustavo& 04] [Baghdadi 12] [Teway& 13]. Les services Web étant accessibles aussi bien à l'intérieur qu'à l'extérieur d'une entreprise, ils permettent à l'entreprise d'intégrer ses applications hétérogènes ainsi que ceux des entreprises partenaires, indépendamment des environnements techniques et des langages sur lesquels tournent ces applications. Cette spécificité présente un atout considérable, favorisant ainsi la mise en place des coopérations B2B. Grâce à la principale caractéristique de couplage faible des services, les apports du paradigme SOA pour la flexibilité des processus métiers sont :

- *Adaptabilité* : les services sont fournis avec un niveau d'abstraction élevé, seules leurs interfaces sont visibles de l'extérieur, leur utilisation dans une application métier est basée sur des interactions faiblement couplées et ne dépend pas de leur implémentation interne. De ce fait, il est plus facile de substituer un service par un autre, ajouter ou supprimer un service, … etc.
- *Evolutivité* : l'encapsulation de la complexité des processus métiers dans des services plus réduits simplifie l'évolution des services eux-mêmes et l'évolution des processus qui les utilisent, ce qui induit donc une maintenabilité et une évolutivité plus aisée.
- *Réutilisabilité* : il est plus facile de réutiliser un service puisqu'il n'est pas directement lié à d'autres services. De plus, la composition de services est un mécanisme qui permet de combiner les services fournis par les partenaires pour offrir d'autres services à forte valeur ajoutée.

Nous proposons une démarche de restructuration des modèles de processus WF à base d'activités en modèles de WF à base de services, en vue de réaliser leur interconnexion via les invocations de services. Les WF, au lieu d'être exprimés avec le concept d'activité, sont restructurés pour être décrits avec le concept de service. Cette restructuration s'avère nécessaire d'une part, dans le cas où les processus métiers des partenaires sont déjà mis en place selon une approche de WF à base d'activités ; et d'autre part pour mettre en évidence la manière de découper les processus des partenaires en services de telle sorte à respecter les contraintes d'abstraction requises pour chaque schéma de coopération à réaliser.

a. *Méta-modèles de worflow*

Cette démarche de restructuration s'appuie sur le méta-modèle de Workflow à base d'activités (Figure 40) et le méta-modèle de Workflow à base de services (Figure 41).

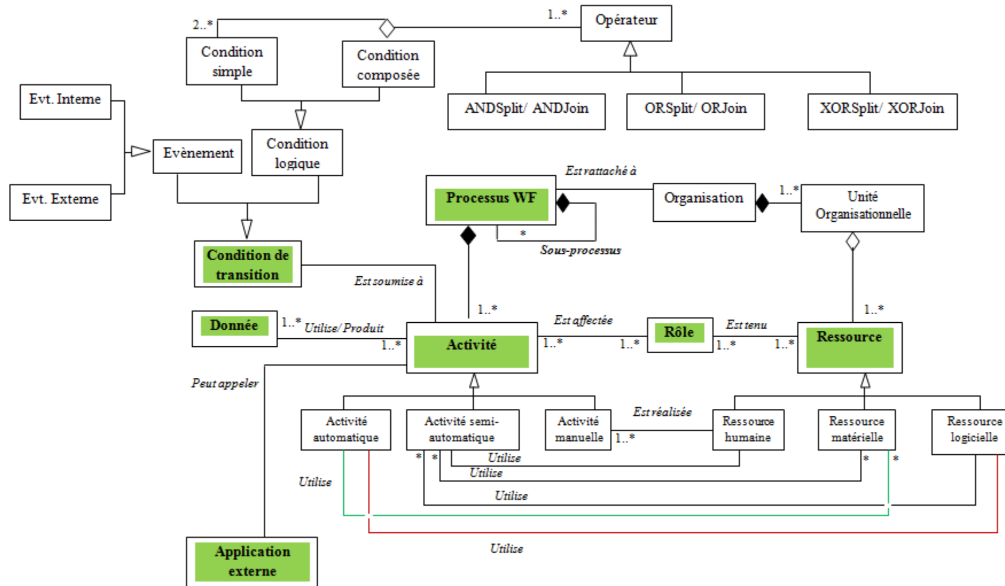


Figure 40 : Méta-modèle de Workflow à base d'activités.

Au niveau intra-organisationnel, un processus de WF est rattaché à une organisation qui est structurée en un ensemble d'unités organisationnelles comportant des ressources humaines, matérielles et logicielles. Le processus WF peut être décomposé en un ensemble de sous-processus qui ont la même structure que le processus global. Au niveau le plus bas de la décomposition, on trouve les activités. Une activité utilise et produit des données, son exécution est soumise à une condition de transition pouvant être un événement interne ou externe à l'organisation, elle peut aussi être une condition logique simple ou composée permettant de contrôler les exécutions parallèles, alternatives et les synchronisations dans le processus. L'activité peut être manuelle réalisée par une ressource humaine, semi-automatique faisant intervenir une ressource humaine et une ressource de type matériel ou logiciel ou automatique réalisée par une ressource matérielle ou logicielle. Une activité peut appeler une application externe (qui n'est pas développée dans le cadre du WF). Le concept de rôle est très important dans le WF car il permet à l'exécution une affectation flexible des ressources selon leurs compétences dans le processus et leur charge de travail. Ainsi, une activité affectée à un rôle donné peut être réalisée par n'importe quelle ressource qui détient ce rôle.

Si l'on considère un partenaire métier représentant une organisation donnée qui implémente son processus métier (ou son WF) selon une approche SOA, on peut dire qu'un processus métier (ou un WF) est défini par une composition de services.

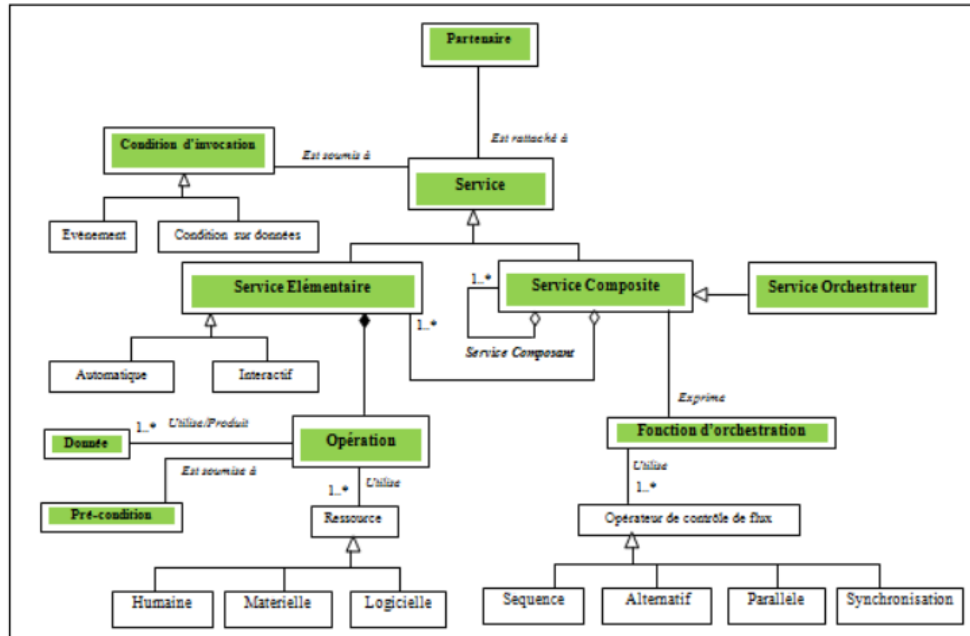


Figure 41 : Méta-modèle de définition de processus WF selon une approche SOA.

Un service peut être composé d'autres services chacun remplissant une sous-fonctionnalité du processus global, on parle de service composant. Particulièrement un service composite peut être le service orchestrateur qui définit le contrôle de flux d'un processus WF composé de services, via une fonction d'orchestration. Celle-ci utilise des opérateurs de contrôle de flux pour exprimer la séquence, le parallélisme, l'exécution alternative et la synchronisation entre services. Au niveau le plus bas de la décomposition, on retrouve les services élémentaires qui remplissent des fonctions élémentaires à travers les opérations qui les implémentent. Une opération est soumise à une pré-condition, produit un effet et génère une post-condition. Chaque opération utilise des ressources pour sa réalisation. Un service (élémentaire ou autre) peut être interactif (semi-automatique) ou automatique. Un service interactif utilise des ressources matérielles et/ou logicielle et nécessite l'intervention d'une ressource humaine contrairement au service automatique qui s'exécute sans une quelconque interaction avec un utilisateur humain.

Comparaison et conclusion : en comparant les deux méta-modèles, à travers la correspondance de concepts établie, nous pouvons dire qu'une *activité* peut être encapsulée dans un *service élémentaire*, un *sous-processus* peut être encapsulé dans un *service composant* qui peut lui-même être *composite* (puisque le sous-processus peut être composé d'autres sous-processus) et un *processus WF* peut être encapsulé dans le *service orchestrateur* qui est le service composite global.

Pour la phase d'interconnexion, nous proposons un ensemble de patrons de coopération que nous avons conceptualisés à l'aide de méta-modèles, accompagnés d'un ensemble de règles de restructuration et d'interconnexion et un ensemble de contraintes liées aux flux de données dans le processus global.

b. Démarche de restructuration et d'interconnexion de processus

La démarche s'articule autour de trois phases principales (voir Figure 42) : une phase de mapping, une phase d'abstraction et une phase d'interconnexion :

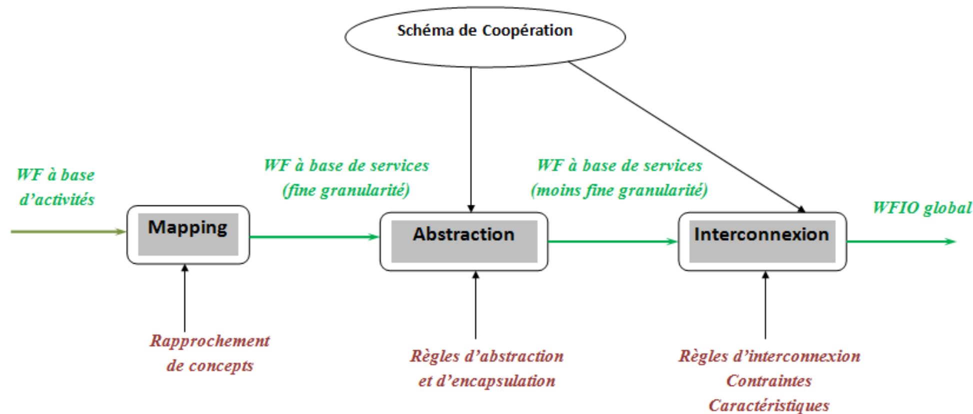


Figure 42 : Démarche générale de restructuration et d'interconnexion de processus.

1. *Phase de mapping :* elle considère que les WF implémentés sont des WF à base d'activités que nous voulons restructurer en WF à base de services. A cet effet, conformément au rapprochement entre les concepts d'activité et de service élémentaire, il s'agit d'identifier les activités du processus et d'encapsuler chaque activité dans un service élémentaire.

2. *Phase d'abstraction :* elle prend en entrée le WF à base de services issu de la première phase. Elle construit un WF à base de services avec une granularité moins fine, de manière à abstraire les détails fonctionnels du processus en (1) identifiant les points d'interaction entre les processus à interconnecter ; en (2) découpant le processus en sous-processus au niveau des points d'interaction et en (3) encapsulant chaque sous-processus dans un service composant.

3. *Phase d'interconnexion :* elle réalise l'interconnexion (ou la composition) selon les règles spécifiques à chaque schéma de coopération. Globalement, un schéma d'interconnexion est proposé pour chaque patron afin de prendre en compte les caractéristiques et les contraintes de chaque architecture de WFIO quant au découpage du WF en services, le contrôle d'exécution des instances et la structure d'interaction (sens des invocations/réponses, les données transmises, les points de synchronisation… etc.).

c. Approche à base de patrons : les PCBS

Nous caractérisons les architectures de base de WFIO par trois dimensions : la *distribution des services*, le *contrôle d'exécution* et la *structure d'interaction*. Nous considérons que les services sont distribués sur les sites des partenaires qui les fournissent, le contrôle d'exécution peut être centralisé, décentralisé ou hiérarchisé et même parfois mixte (une combinaison des modes précédents). Les interactions peuvent être de type synchrone ou asynchrone. Nous introduisons le concept de « *patron de coopération à base de services » (PCBS*) :

> *Un PCBS définit la manière dont les services sont répartis sur les sites des partenaires, la structure d'interaction entre ces services et leur contrôle d'exécution, donnant lieu soit à une orchestration soit à une chorégraphie de services.*

Notre approche d'interconnexion de WF selon un schéma de coopération donné repose sur trois questions essentielles :

1. Comment restructurer les processus WF en services ?
2. Comment réaliser le contrôle d'exécution des instances ?
3. Comment définir les interactions entre les services fournis par les différents partenaires ?

Ces trois questions définissent les trois dimensions principales sur lesquelles repose notre définition du concept de patron de coopération à base de services (PCBS). Par suite, un PCBS est défini par : la *distribution des services* sur les sites des partenaires, le *contrôle d'exécution* et la *structure d'interaction* entre les différents fragments de processus.

Voici le méta-modèle d'un PCBS (ou *SBCP* pour *Service-Based Cooperation Pattern*) :

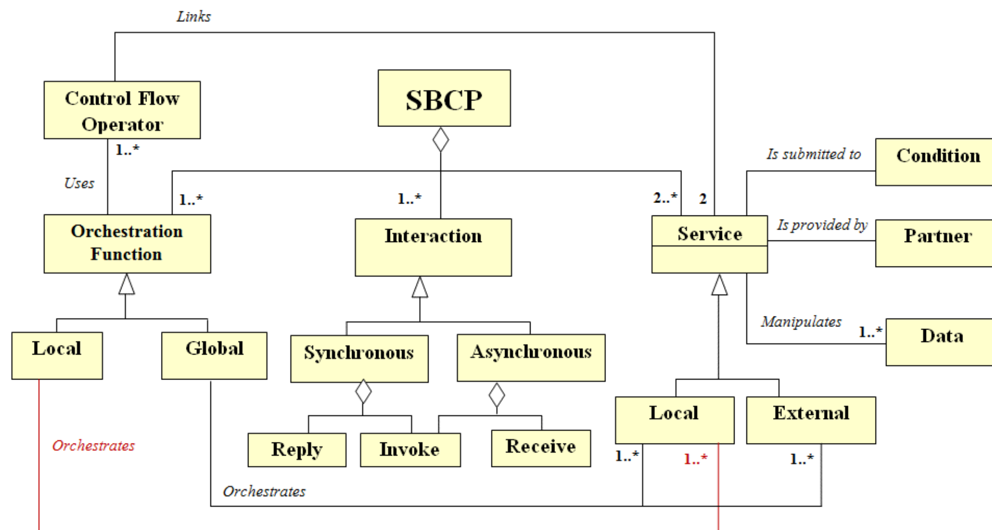


Figure 43 : Méta-modèle d'un Patron de Coopération à Base de Services (PCBS).

Ce méta-modèle représente la base commune pour décrire les différents PCBS proposés dans le cadre de ces travaux. Nous avons proposé six patrons de coopération : *Partage de charge*, *Exécution chainée*, *Sous-traitance*, *Transfert de cas*, *Transfert de cas étendu* et *Faiblement couplé*.

La conceptualisation d'un patron repose sur un méta-modèle de définition de WFIO obéissant au patron en question que nous avons obtenu à chaque fois par adaptation et raffinement du méta-modèle générique (Figure 43). La conceptualisation consiste aussi à spécifier les contraintes sur le flux de données dans le processus global, en plus des caractéristiques de l'opération d'interconnexion (composition proprement dite ou interconnexion de modèles) pour chaque patron de coopération. Conceptuellement, chaque schéma de coopération correspondant à une architecture donnée de WFIO et est supporté par un PCBS définissant les règles de distribution de services, de contrôle d'exécution et d'interaction entre ces services qui diffèrent d'un schéma de coopération à l'autre. Ainsi, pour chaque patron de coopération, les informations suivantes sont fournies : une description générale comportant le nom, la référence, le cas d'utilisation, un schéma générique du patron, un méta-modèle, un ensemble de règles d'abstraction et un ensemble de règles et contraintes d'interconnexion. Celles-ci sont fournies pour le guidage de l'implémentation du patron.

Voici, à titre illustratif un des six patrons proposés : le patron « Partage de Charge » - PCBS1 :

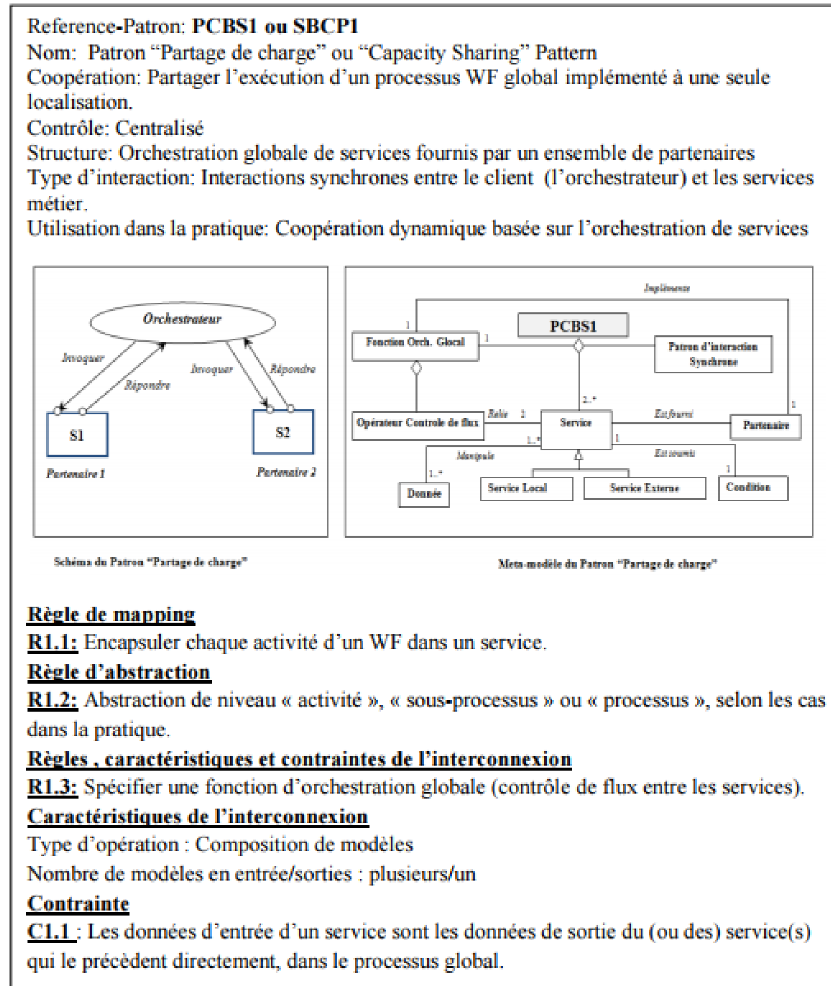

Reference-Patron: **PCBS1 ou SBCP1**
Nom: Patron "Partage de charge" ou "Capacity Sharing" Pattern
Coopération: Partager l'exécution d'un processus WF global implémenté à une seule localisation.
Contrôle: Centralisé
Structure: Orchestration globale de services fournis par un ensemble de partenaires
Type d'interaction: Interactions synchrones entre le client (l'orchestrateur) et les services métier.
Utilisation dans la pratique: Coopération dynamique basée sur l'orchestration de services

Schéma du Patron "Partage de charge"

Meta-modèle du Patron "Partage de charge"

Règle de mapping
R1.1: Encapsuler chaque activité d'un WF dans un service.
Règle d'abstraction
R1.2: Abstraction de niveau « activité », « sous-processus » ou « processus », selon les cas dans la pratique.
Règles , caractéristiques et contraintes de l'interconnexion
R1.3: Spécifier une fonction d'orchestration globale (contrôle de flux entre les services).
Caractéristiques de l'interconnexion
Type d'opération : Composition de modèles
Nombre de modèles en entrée/sorties : plusieurs/un
Contrainte
C1.1 : Les données d'entrée d'un service sont les données de sortie du (ou des) service(s) qui le précèdent directement, dans le processus global.

Figure 44 : Illustration des PCBS : patron « Partage de Charge » - PCBS1

2.5. Support de la flexibilité des modèles de WFIO

Nous considérons les aspects de flexibilité de WFIO selon trois aspects : *adaptabilité*, *évolutivité* et *réutilisabilité*. Nous introduisons des concepts et des opérateurs spécifiques support de l'adaptation, l'évolution et la réutilisation des modèles de processus WFIO. Nous proposons des patrons d'adaptation/évolution pour la prise en charge des changements de différents types pouvant affecter les modèles de processus WFIO obéissant aux différents patrons de coopération que nous avons définis.

La réutilisabilité d'un modèle de WFIO définit sa capacité à être intégré à d'autres modèles et manipulé comme une entité à part entière, afin de construire des modèles de WFIO plus complexes. Contrairement à la réutilisabilité, l'*adaptabilité* et l'*évolutivité* ont introduit de nouveaux concepts que nous détaillons.

a. Adaptabilité et adaptation d'un modèle de WFIO

L'adaptabilité d'un modèle définit sa capacité à supporter aisément des modifications, ajouts ou suppressions au niveau des entités qui le composent sans altérer la cohérence et le fonctionnement global du processus et sans impact sur l'aspect coopération. Nous avons proposé et implémenté des

patrons d'adaptation classés en trois catégories selon les trois dimensions de définition d'un PCBS : patrons d'adaptation de services, patrons d'adaptation de contrôle de flux et patrons d'adaptation des interactions. Des opérateurs spécifiques d'adaptation ont été définis pour la formalisation des patrons d'adaptation de base.

> *Un patron d'adaptation (PA) décrit les opérations à effectuer afin d'apporter une modification bien définie sur un modèle de WFIO source pour arriver à un modèle de WFIO cible, tout en maintenant la cohérence du modèle global.*

L'adaptation se décline en trois catégories : adaptation de services (ajout, suppression, substitution, fusion et décomposition) ; adaptation du contrôle de flux (avec des patrons de séquentialisation de services et des patrons d'alternation ou de parallélisation de services) ; adaptation des interactions par des adaptation pour chaque type de patron.

b. Evolutivité et Evolution des modèles de WFIO

L'évolutivité d'un modèle de WFIO désigne sa capacité à supporter une expansion de sa fonctionnalité globale et/ou de sa coopération (ouverture du processus métier à de nouveaux partenaires). Notons que les deux perspectives ne sont pas exclusives, l'expansion de la coopération induit forcément une expansion de la fonctionnalité globale. Nous considérons que pour une expansion des fonctionnalités, il suffit d'utiliser les patrons d'adaptation de services. Cependant pour une expansion de la coopération, nous avons introduit deux nouveaux concepts : les patrons de coopération généralisés et les patrons de coopération composites. Notons que l'utilisation de patrons de coopération généralisés ou composites constitue un aspect de réutilisation des modèles de processus WFIO.

> *Un patron de coopération généralisé est un patron de coopération de base (PCBS) appliqué à l'interconnexion de trois processus ou plus.*

> *Un patron de coopération composite est un patron de coopération qui définit l'interconnexion entre deux WFIO déjà existants obéissant à deux PCBS différents.*

Les patrons de coopération généralisés sont des expansions de chaque PCBS. Quant à l'évolution par réutilisation de modèles, nous avons définis deux patrons composites : « Exécution chainée-Sous-traitance » et « Partage de charge-Transfert de cas ».

2.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Thèse de S. Boukhedouma dans le cadre d'un projet PAI Tassili entre l'Université de Nantes et l'USTHB d'Alger.

- Taux d'encadrement : 20%[17].

2.7. Brève synthèse des travaux fondateurs

Tous les travaux que nous avons étudiés sont destinées à une coopération dynamique où les partenaires métiers sont découverts dans un annuaire de publication de services. Ceci met en évidence l'apport de l'architecture SOA dans la coopération B2B dynamique (occasionnelle ou spontanée). Toutes ces approches présentent déjà une flexibilité dans la sélection des services et offrent des mécanismes pour l'adaptation du processus global. Par exemple, e-Flow [Casati& 01] offre des

[17] 20% au lieu de 30% en raison d'une interruption de travail d'un an.

mécanismes pour l'adaptation du modèle global en fonction d'un protocole prédéfini, l'approche Self-Serv [Sheng& 02] offre une adaptation par substitution de services, PYROS [Belhajjame& 05] permet une adaptation du contrôle de flux et une personnalisation des services selon les besoins des utilisateurs. Les approches FOCAS [Pedraza 09] et Orchestration Multi Contextuelle [Esper 10] utilisent le concept de procédé (ou service) abstrait pour supporter l'adaptation de l'application, Astro-CaptEvo [Heorhi 12] offre des mécanismes pour l'adaptation des processus métiers au contexte. Les aspects d'évolution peuvent être déduits de l'association de nouveaux partenaires (ajout de nouveaux services par exemple, lors d'une adaptation du processus). Quant aux aspects de réutilisation, ils concernent la réutilisation de services basiques dans toutes les approches, la réutilisation de patrons de processus dans e-Flow, la réutilisation de WF dans l'approche CoopFlow [Chebbi 07] et la réutilisation de services composites dans Self-Serv.

Outre les nouveaux concepts et les formalisations que nous introduisons, notre approche d'interconnexion diffère des approches existantes par le fait que les processus métiers que nous considérons ne sont pas appréhendés comme des « boites noires » mais doivent dans certains cas (selon le schéma de coopération à réaliser), exhiber certains détails nécessaires à l'interconnexion appelés *points d'interaction*. La règle générale que nous adoptons est de présenter les modèles de processus WF avec le maximum d'abstraction et le minimum de visibilité requise pour l'interconnexion.

2.8. Positionnement et bilan

Pour répondre au premier verrou, nous avons proposé une démarche de restructuration et d'interconnexion de modèles de WF basée sur le paradigme SOA (Architecture Orientée Service) ; nous avons introduit un nouveau concept appelé Patron de Coopération à Base de Services (PCBS) permettant de caractériser un modèle de processus WFIO à base de services selon trois dimensions principales. Aussi, nous avons procédé à une conceptualisation des différents patrons de coopération proposés et une formalisation de chacun d'eux en définissant un opérateur de coopération approprié. Quant au second verrou, nous percevons la flexibilité des modèles de WFIO obéissant aux nouveaux patrons de coopération définis, sous trois aspects complémentaires : *adaptabilité*, *évolutivité* et *réutilisabilité*. A cet effet, nous avons proposé et formalisé à l'aide d'opérateurs dédiés, un ensemble de patrons d'adaptation. Pour les aspects d'évolution et de réutilisation de modèles, nous avons introduit les concepts de patron de coopération généralisé et patron de coopération composite qui décrivent un aspect de réutilisation des modèles de WFIO, pour la construction de nouveaux modèles de WFIO plus complexes. Nous avons ainsi proposé d'encapsuler des workflows d'entreprises au sein de services web en adoptant une approche SOA afin de faciliter l'interconnexion de ces workflows tout en identifiant des patterns d'interconnexion et gérer leur évolution.

Ma contribution : ma contribution a porté essentiellement sur l'encadrement sur les patrons d'adaptation et d'évolution. A noter cependant que durant la thèse de S. Boukhedouma, j'ai eu une interruption de travail pendant 1 an (2013), d'où mon taux d'encadrement à 20% au lieu des 30% initialement prévus.

Validation : différents outils ont été développés, notamment au travers de stages d'étudiants en cycle ingénieurs, étudiants de S. Boukhedoua qui était déjà en poste à l'Université d'Alger. Les frameworks ont été développés sous la plateforme Windows 7. BPEL a été le langage de spécification de nos processus WF devant être interprétés et exécutés par le moteur de WF Open-ESB2.2 et le serveur GlassFich. Les EJB (Enterprise Java Beans) ont servi au développement des services Web, et langage java, l'IDE Net-beans et l'API Jdom ont servi au développement des applications.

Valorisation : ces travaux ont donné lieu à 10 publications, notamment internationales, dont 2 revues internationales International Journal of Business Information Systems 2015 et International Journal of Information Technology and Computer Science 2014 (références [4] [5] du CV détaillé), 1 chapitre d'ouvrage (référence [12] du CV détaillé), 2 conférences de rang B IEEE RCIS 2013 et BIS 2012 (références [39][40] du CV détaillé), 3 autres conférences internationales et 2 conférences nationales.

3. Externalisation du modèle SaaS mutualisé et de sa gestion

3.1. Fiche d'identité

Titre :	Une contribution à la gestion des applications SaaS mutualisé dans le Cloud : approche par externalisation
Paradigme(s) :	Cloud – SaaS mutualisé – SOA
Principales contributions :	. Externalisation de la gestion de la variabilité dans les applications SaaS mutualisé. Spécification d'un méta-modèle de variabilité. Proposition de la notion de *VaaS* (*Variability as a Service*).
Mots-clés :	Cloud Computing, SaaS, mutualisation, multi-locataires, externalisaion, architecture orientée service, variabilité, méta-modèle.
Collaborateur(s) :	A. Ghaddar (doctorant), A. Assaf (encadrant entreprise, BITASOFT) S. Dupont (Master 2) – Encadrement : 100%.
Mon implication :	Encadrement à 100% sur le plan scientifique.
Cadre de collaboration :	Thèse de A. Ghaddar – Thèse CIFRE
Période :	2010 – 2013
Principales publications :	1 conférence de rang A : Caise 2012 ; 1 conférence de rang B : IEEE SOSE 2011 ; 1 conférence nationale : INFORSID 2012.

Figure 45 : Fiche synthétique.

3.2. Synthèse

La multi-location (*multitenancy*) est un principe d'architecture logicielle relativement nouveau et faisant partie du modèle SPI (*Software-Platform-Infrastructure*) du Cloud (Chapitre 4. 1. d). Ce principe est généralement adopté lorsqu'une application est fournie sous forme d'un service : on parle alors de mode *SaaS* (*Software as a Service*) (Chapitre 4. 1. e). Ce principe réduit considérablement les coûts de déploiement et de maintenance de l'application ainsi que des économies d'échelle. La raison est que tous les utilisateurs (*locataires* ou *tenants*) partagent la même instance de cette application. Les principaux bénéfices sont l'automatisation de la mise en place de locataires, la réduction de la complexité opérationnelle et l'optimisation des ressources infrastructurelles. Toutefois, pour toucher un grand nombre de locataires, l'application doit être personnalisable et configurable pour répondre aux exigences variables de chaque locataire. Les fournisseurs SaaS font face à une préoccupation supplémentaire car la mise en œuvre de la multi-location nécessite de relever un certain nombre de défis liés à sa structure organisationnelle, au sein de laquelle chaque locataire doit avoir l'impression d'utiliser une application qui lui est pleinement dédiée. Cela implique une gestion dynamique de la variabilité des besoins de locataires et une isolation stricte de leurs données.

Nous prônons l'externalisation de la gestion de la variabilité. Nos travaux s'inscrivent ainsi dans la tendance vers une gestion globale d'externalisation sous forme de service, connu sous l'acronyme *XaaS* (*everything as a Service*) [Rimal& 09]. Nos travaux s'inscrivent dans les niveaux 3 et 4 de maturité du SaaS mutualisé (Chapitre 4. 1. e), ces niveaux offrant un plus haut degré de mutualisation étant donné qu'une seule instance de l'application est proposée et maintenue. Nos contributions se résument en deux axes : (i) le premier concerne la spécification d'un méta-modèle de variabilité pour traiter la variabilité et externaliser sa gestion sous forme d'un service : *VaaS (Variability as a Service)*. (ii) Le second axe consiste à étendre la politique de gestion par externalisation au niveau des données en proposant un système d'isolation de données sous forme d'un service *DIaaS (Data Isolation as a*

Service). Le principal avantage de ce système est d'isoler les données de locataires d'une manière quasi-transparente aux développeurs, sans introduire de changements majeurs sur les architectures des applications existantes.

3.3. Verrous scientifiques

Malgré ses avantages indéniables de réduction des coûts et des efforts de maintenance, la mutualisation du SaaS présente une série de défis conceptuels et techniques [Krebs& 12] [Guo& 07] [Pervez& 10]. Les principaux défis sont la gestion de la variabilité, l'isolation des données de locataires, le contrôle d'accès aux ressources de l'application, la maintenance [Bezemer& 10] [Koziolek 11]. Nous nous sommes adressés aux deux premiers verrous :

- *La gestion de la variabilité :* sa complexité réside dans le fait qu'elle affecte tous les aspects de l'application, de l'aspect fonctionnel tel que l'apparence et le processus métier, jusqu'à l'aspect non-fonctionnel concernant par exemple la qualité de service. Cette gestion a été largement traitée dans l'état de l'art en tant que problématique incontournable à surmonter absolument dans ce contexte [Mietzner& 08], [Mietzner& 09], [Kabbedijk& 11] [Schroeter& 12]. La gestion de la variabilité est généralement intriquée au sein du logiciel lui-même.
- *L'isolation des données de locataires :* dans une application mutualisée, les données de tous les locataires sont consolidées dans une instance de base de données unique. De ce fait, il est primordial de mettre en place un système d'isolation de données capable de garantir qu'un locataire n'aura jamais le privilège d'accéder aux données d'autres locataires (potentiellement des concurrents) sauf si ce privilège a été explicitement accordé [Wang& 08]. Cette deuxième contribution n'ayant pas encore été publiée, elle ne sera pas détaillée dans le présent document.

Avant nos contributions, précisons le contexte et les constats que nous faisons :

3.4. Gestion de la Variabilité des besoins de locataires

Le principe de mutualisation de SaaS reste encore flou. En nous basons sur différentes définitions du SaaS mutualisé [Bezemer& 10] [Weissman& 09] [Krebs& 12], nous proposons la suivante :

> *Une application SaaS mutualisée consiste à faire partager ses ressources entre plusieurs locataires, dans le but de réduire sa complexité opérationnelle et de tirer pleinement parti des économies d'échelle. L'application est potentiellement déployée sur plusieurs serveurs afin de réagir efficacement contre les variations de charges et d'optimiser l'utilisation des ressources infrastructurelles. Un locataire de cette application peut la configurer et la reconfigurer à n'importe quel moment pour l'adapter à ses besoins. Sa configuration et ainsi ses données sont entièrement isolées de tout impact par d'autres locataires.*

a. Le style architectural du modèle SaaS mutualisé

Koziolek [Koziolek 10] [Koziolek 11] propose un style architectural définissant les éléments fondamentaux de toute architecture des applications SaaS mutualisées. La majorité des éléments architecturaux figurant dans ce style (composants, connecteurs, données, etc.) présentent une extension de ceux qui existent dans des styles architecturaux classiques (Client/Serveur, pipe-and-filter, multi-couches, REST, etc.).

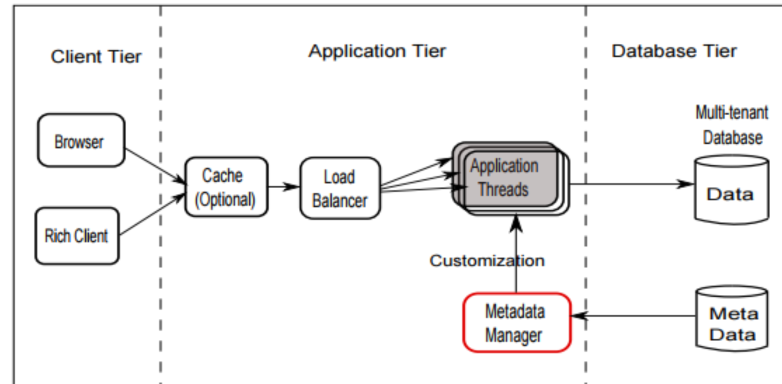


Figure 46 : Un style architectural pour les applications SaaS mutualisées [Koziolek 10].

La nouveauté de ce style consiste à introduire un élément architectural additionnel : celui de la *gestion de méta-données*. Son rôle est de fournir les informations nécessaires à l'adaptation dynamique de l'application dans ses différents aspects (interfaces graphiques, processus métiers, services, etc.). Elle évite le codage en dur des fonctionnalités variables et permettent leurs adaptations en temps réel et de manière dynamique [Li Heng& 12] [Chong& 06] [Kwok& 08] [Namjoshi& 09] [Weissman& 09].

b. Point sur la gestion de la variabilité

D'après notre analyse de l'état de l'art concernant ce défi [Sengupta& 11] [Mietzner& 09] [Sun& 10][Mietzner& 08a][Kabbedijk& 11], nous avons pu identifier trois activités de gestion principales :

1. *La séparation entre la commonalité et la variabilité :* en identifiant les parties de l'application qui sont communes entres les locataires et les autres parties qui varient d'un locataire à un autre. L'approche adoptée et adaptée est celle des lignes de produits (LDP), notamment au travers des travaux de Miertzner [Mietzner& 09].
2. *La modélisation de la variabilité :* la modélisation explicite de la variabilité lors de sa gestion permet d'améliorer la prise de décisions en forçant les développeurs à documenter et à justifier les raisons qui les poussent à l'introduire. Elle permet également d'améliorer la communication entre les différents acteurs de l'application en fournissant une abstraction des artefacts variables sans nécessité de rentrer dans les détails techniques [Pohl& 05]. La variabilité, selon les LDP, est principalement documentée à travers :
    a. *Points de variations* identifiant précisément dans l'architecture où des changements/variations peuvent apparaître. Un point de variation concrétise une décision de conception retardée à un stade ultérieur, dans lequel une idée plus claire sur le contexte du client pourra être connue [Jacobson& 97].
    b. *Variantes* : caractérisent un ou plusieurs choix d'un point de variation. Elles représentent des choix architecturaux possibles sur le point de variation. Lorsqu'un choix architectural est effectué et implémenté, le point de variation devient résolu.
    c. *L'intégration des informations de la variabilité dans les artefacts :* fait que la variabilité est dispersée à travers différents artefacts. Cela augmente la taille et la complexité du modèle de variabilité.
    d. *Orthogonalité de définition :* conduit par la nécessité de décrire les dépendances de variabilité, certaines approches ont été proposées suggérant de définir la variabilité dans un modèle orthogonal. Le plus connu est le modèle OVM (Orthogonal Variability Model)

[Pohl& 05] [Metzger& 07]. Il définit ainsi les concepts de *Point de variation (PV)*, *variante* et *dépendance de variabilité*. Les concepts *Artefacts de développement* et *dépendance d'artefacts* établissent des liens de traçabilité entre les éléments de la variabilité (en termes de PV et des variantes) et les artefacts de développement concernés. Les deux derniers concepts sont : *Contraintes*, qui définissent des règles de dépendance, et *Variations interne et externe*, pour identifier respectivement les PV visibles uniquement aux développeurs et ceux communiqués aux utilisateurs.

3. *La résolution de la variabilité :* en adaptant dynamiquement l'architecture de l'application mutualisée en fonction de son modèle de variabilité et les configurations de chaque locataire :
   a. Les principales approches sont celles des *lignes de produits dynamiques (LDPD)*. Les approches existantes [Trinidad& 07] [Bencomo& 08] [Cetina& 08], [Dinkelaker& 10] ont particulièrement mis l'accent sur l'adaptation dynamique des systèmes à un contexte d'utilisation spécifique. Toutefois elles ont certains inconvénients dont celui de ne pas supporter le concept de locataire. Il existe également des cadres de développement orientés aspects [Truyen& 01] [Lagaisse& 06] qui supportent un tissage dynamique et distribué d'aspects. Ces cadres sont donc adaptés à une utilisation dans un contexte SaaS mutualisé. Cependant, ils sont tous conçus pour une utilisation dans des architectures d'applications à base de composants, et ne supportent pas les applications construites à base de services.
   b. Les approches existantes d'adaptation dynamique des services sur les plans conceptuel et technique, à l'instar des approches de LDPD, peuvent être utilisées pour adapter l'architecture orientée services d'une application SaaS mutualisée [Guo& 07] [Laplante& 08] [Mietzner& 09]. Ainsi, les mécanismes existants de substitution ou de changement de l'ordre d'exécution dynamique de services restent valides et utiles mais le temps de résolution de ces points, dans leur majorité, sera retardé jusqu'au moment de l'exécution.

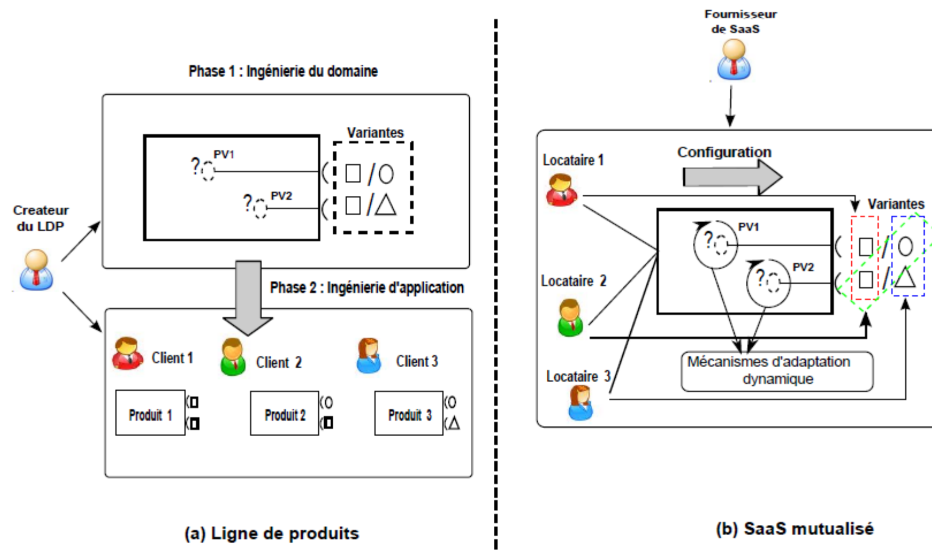


Figure 47 : La variabilité dans les LDP Vs. la variabilité dans les applications SaaS mutualisées

c. Constat et approche suivie

Constat : nous avons rencontré certaines limites d'OVM pour modéliser des nouvelles situations de variabilité qui sont principalement liées à la différence entre la structure organisationnelle d'une

application SaaS mutualisé et celle d'une LDP (Figure 47). Dans [Bosch 01] [Pohl& 05], différentes structures organisationnelles pour les LDP ont été étudiées et évaluées. Toutes ces structures partagent le fait que l'ingénierie d'une application doit être uniquement réalisée par la même organisation qui a réalisé l'ingénierie du domaine. Cette contrainte est due aux mécanismes de dérivation fournis par les approches de LDP, qui nécessitent généralement une intervention humaine pour assembler les composants de chaque nouveau produit et ensuite de le déployer dans l'environnement d'exécution du client. Cela est différent de la structure organisationnelle du modèle SaaS mutualisé qui sépare entre la construction du logiciel effectuée par son fournisseur d'une part, et sa configuration effectuée directement par les locataires à n'importe quel moment d'autre part. De ce fait, des préoccupations supplémentaires doivent être gérées à travers une approche de modélisation telles que le contrôle d'accès au modèle de variabilité (par exemple, l'interdiction de sélection de certaines variantes par les locataires non autorisés) et son auto-description pour permettre sa manipulation directement par les locataires.

Approche suivie : pour couvrir ces limites d'OVM, nous introduisons un nouveau modèle de variabilité orthogonale (Figure 48). Ce modèle étend OVM avec de nouveaux concepts de modélisation lui permettant de répondre aux besoins émergents des applications SaaS mutualisées, tout en conservant la même méthodologie de réflexion sur la modélisation de la variabilité principalement établie par OVM.

L'une des principales limitations des approches existantes est celle de leur fragmentation : la majorité visent uniquement des problèmes spécifiques propres à un aspect particulier et sur une couche particulière de la SOA, tandis que les relations entre les couches sont ignorées par l'isolation des solutions et leur diversité. En conséquence, cela pourra provoquer un manque d'uniformité qui risque de compliquer la réflexion sur la variabilité et de conduire à la prise de mauvaises décisions où les modifications sur une couche pouvant endommager les fonctionnalités d'une autre couche. Au lieu de travailler sur une couche particulière en lui cherchant une solution de variabilité « ultime », nous essayons de nous focaliser sur la proposition d'une solution globale. Cette solution considère deux activités de gestion essentielles : la *modélisation* et la *résolution de la variabilité*.

3.5. Modélisation de la variabilité : nos contributions

a. Notre méta-modèle de variabilité : concepts et formalisations

Notre méta-modèle de la variabilité (Figure 48) est instancié par un fournisseur de service (SaaS ou service SOA) et peut être composé avec d'autres modèles de variabilité appartenant à d'autres services. Son instanciation doit être suivie par la définition de l'ensemble des points de variation qui lui appartiennent. Ces derniers représentent les éléments principaux du modèle et sont attachés à des rôles pour permettre à plusieurs personnes d'intervenir à la configuration de l'application. Un point de variation définit également un ensemble de variantes qui modélisent les différentes options possibles sur ce point. Celles-ci sont associées à des artefacts de développement qui documentent les éléments architecturaux qui les réalisent. Une classe valeur de cette association (variante-artefact) est introduite pour saisir les informations nécessaires à la résolution de la variabilité. Chaque valeur est soit prédéfinie par le fournisseur, soit libre à saisir directement par les locataires.

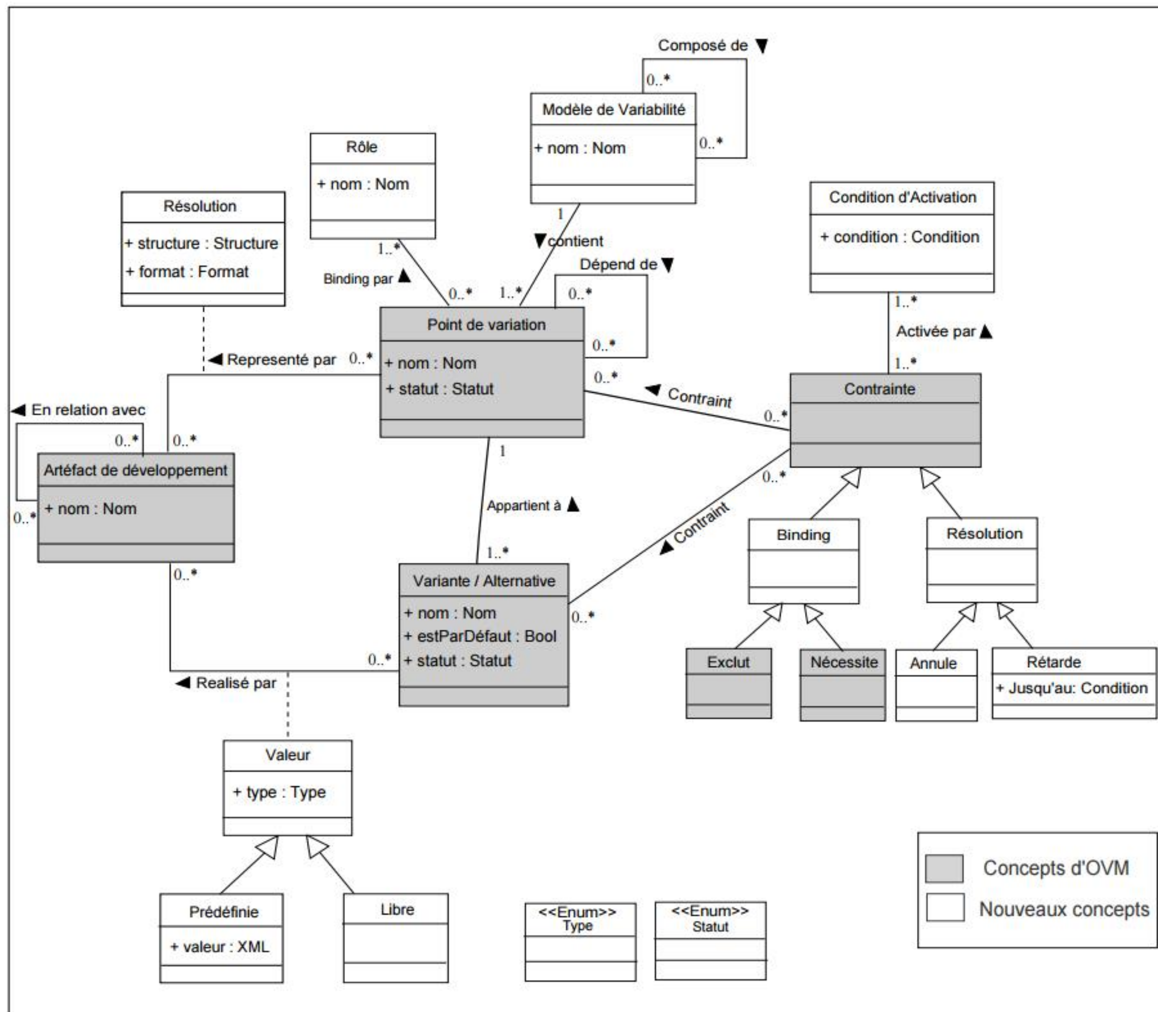


Figure 48 : Notre méta-modèle de variabilité.

La Figure 49 montre un exemple d'instanciation du méta-modèle avec l'utilisation de conditions d'activation. Dans cet exemple, nous pouvons remarquer que les contraintes ne sont plus statiquement définies entre les variantes et les points de variation. Elles sont attachées à des conditions d'activation qui sont évaluées à travers les expressions de leurs conditions. La définition des conditions de base, tel que nous l'avons évoqué, s'opère sur l'utilisation d'un ensemble de fonctions prédéfinies par notre cadre de modélisation.

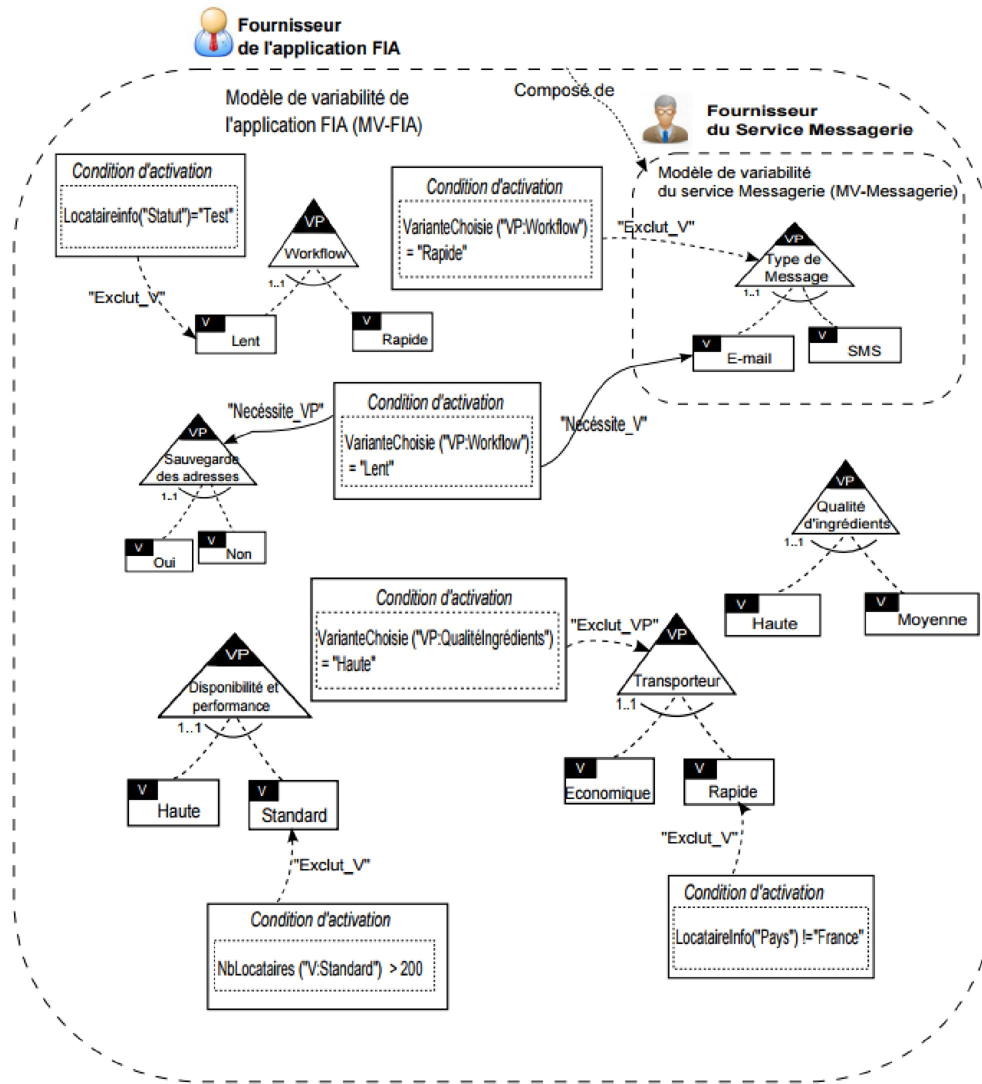


Figure 49 : Exemple d'instanciation du méta-modèle.

b. Variabilité sous forme d'un service : l'architecture de VaaS et le processus d'externalisation de la gestion de la variabilité

Notre approche propose de considérer la gestion de la variabilité sous forme d'un service. Le système de gestion de la variabilité sous forme d'un service (*VaaS : Variability as a Service*) implique la nécessité d'un nouveau type de fournisseur de service : le fournisseur de VaaS. Ce dernier propose à ses clients (c.à.d les fournisseurs de SaaS) de modéliser et de résoudre la variabilité de leurs applications sur sa propre infrastructure.

Le système VaaS fournit un ensemble de composants logiciels génériques dédiés à la gestion de la variabilité des applications mutualisées, indépendamment de leurs technologies de développement utilisées. Ces composants permettent aux fournisseurs de SaaS (ou même les fournisseurs des services Web classiques) de gérer la variabilité des applications sur l'infrastructure du système proposé. Cela revient à modéliser dans un premier temps cette variabilité suivant les concepts de modélisation présentés, et ensuite de la résoudre à travers l'envoi de demandes de résolution en temps réel. L'architecture du système VaaS est présentée dans la Figure 50. VaaS défini un ensemble d'étapes à

suivre par les fournisseurs de SaaS et par les locataires dans le but d'externaliser la gestion de la variabilité. Ces étapes sont divisées en deux catégories principales : la *spécification* et la *résolution*.

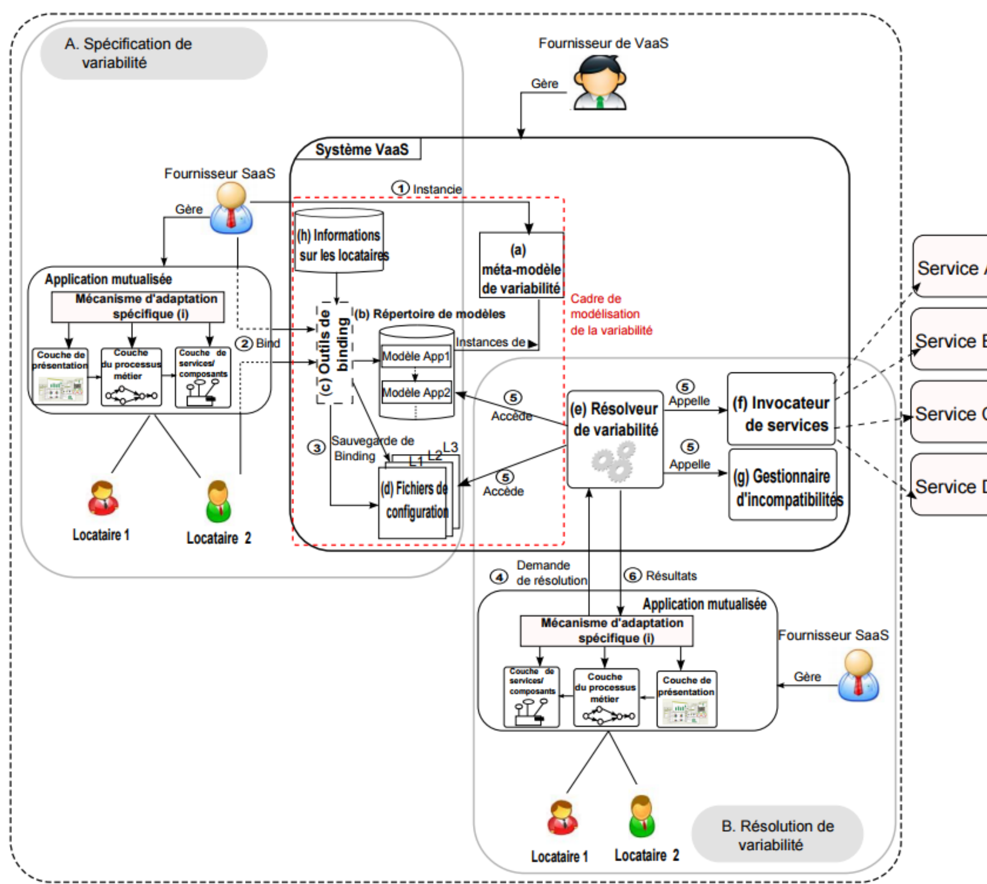


Figure 50 : Architecture de VaaS.

1. *Étapes de spécification :* ces étapes consistent principalement à modéliser correctement la variabilité de l'application et à la configurer complètement par les locataires (Figure 50.A). Pour modéliser la variabilité, le fournisseur de SaaS doit tout d'abord instancier (1) le méta-modèle de variabilité (a). Le modèle de variabilité résultant de cette instanciation sera ensuite sauvegardé dans un répertoire spécifique (b) contenant tous les modèles de variabilité appartenant à toutes les applications clientes du système. Pour différencier ces modèles, chacun sera associé au nom de l'application qu'il représente. Une fois la variabilité modélisée, le binding de celle-ci peut commencer (2). Celui-ci est effectué par le biais d'un outil de binding (c). Une fois le binding effectué, les variantes choisies par chaque locataire seront sauvegardées (3) dans un fichier de configuration (d) qui porte le nom de ce locataire et stocke une représentation sérialisée de cette configuration sous forme d'un fichier XML. À ce moment, ce locataire peut commencer à utiliser l'application et s'attendre à une fonctionnalité et une qualité de service équivalentes à la configuration effectuée.
2. *Étapes de résolution :* ces étapes consistent à adapter l'application à l'exécution, à travers des demandes de résolution à envoyer en temps réel par l'application. Pour chaque endroit variable intercepté, une demande de résolution sera envoyée au système VaaS (4) visant le point de variation intercepté. Lorsqu'une demande est reçue, elle sera traitée par un résolveur de variabilité (d) qui accède au modèle de variabilité de l'application et aux fichiers de configuration du

locataire pour identifier l'action de résolution appropriée à effectuer (5). Les résultats de résolution seront retournés à l'application pour pouvoir continuer son exécution (6). Les méta-données définissant les structures de données requises par l'application suite à la résolution de chaque point de variation doivent être explicitement définies dans le modèle de variabilité, et ce à travers l'attribut structure de la classe d'association Résolution entre le point de variation à résoudre et l'artefact de développement concerné par cette résolution.

3.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Encadrement doctoral (confère *section 7. Détails de l'encadrement doctoral* du CV détaillé) :

1. **2010-2013 :** thèse de Ali GHADDAR (Doctorat en informatique de l'Université de Nantes), Une contribution à la gestion des applications SaaS mutualisées dans le cloud : approche par externalisation »

   - Taux d'encadrement scientifique : 100%.

2. **2012** : Simon DUPONT, Master 2 de l'Université de Nantes : « Orchestration et composition dynamique de la variabilité de services pour des applications Web multi-locataires »
   - Taux d'encadrement : 100%.

3.7. Brève synthèse des travaux fondateurs

D'après l'analyse que nous avons faite des approches et techniques proposées pour implémenter la variabilité, nous pouvons remarquer qu'elles sont différentes ainsi que les circonstances dans lesquelles elles pourraient être appliquées. Cela peut conduire à une forte probabilité d'avoir, dans la même application, de nombreuses solutions de représentation de la variabilité, dont chacune est adaptée à un problème spécifique et qui plus est pour une couche spécifique. Cette diversité de solutions provoque une isolation et un manque d'uniformité, souvent inconscientes, de la modélisation de la variabilité, ce qui complique sa réflexion et sa gestion. Un autre problème complique la gestion : selon notre expérience dans le développement des applications multi-locataire, les développeurs reçoivent habituellement les variations dans les exigences des locataires d'une manière informelle, et elles sont généralement décrites selon leurs expériences et leur propre terminologie du domaine métier. Le manque de formalisation et l'absence d'un modèle uniforme et global de la variabilité fait de la décision pour choisir les techniques et les couches les plus appropriées pour implémenter les variations une tache non-triviale et particulièrement risquée.

Concernant la gestion de la variabilité dans le contexte des systèmes SOA, plusieurs auteurs ont étudié la préoccupation de la variabilité SOA. Parmi les travaux les plus proches aux nôtres, ceux de Dans [Chang& 07] les auteurs identifient quatre types de variabilités qui peuvent se produire dans les systèmes orientés services. Dans [Sun& 10] les auteurs présentent un cadre et des outils pour modéliser et implémenter la variabilité dans les services Web. Dans [Koning& 09] la variabilité est simplement implémentée dans les processus métiers en proposant un langage VxBPEL (une extension du BPEL). Cependant, toutes ces approches se concentrent sur la variabilité dans les systèmes SOA, mais ils ne traitent pas la variabilité dans le contexte d'une application multi-locataire.

Concernant le SaaS et la multi-location, dans [Jansen& 10] les auteurs fournissent un catalogue des techniques de personnalisation qui peuvent guider les développeurs dans le développement des applications multi-locataire. Dans [Bezemer& 10] les auteurs discutent leurs expériences dans la réingénierie des applications à locataire unique vers des applications multi-locataire. Dans [Mietzner& 09], les auteurs proposent une technique de modélisation de la variabilité dans les

applications multi-locataires, ils différencient entre la variabilité interne seulement visible pour les développeurs, et la variabilité externe qui est communiquée aux locataires de l'application. Cependant, ils ne tiennent pas compte de la gestion de la variabilité à travers les différentes couches de l'application.

Notre principale contribution a été de représenter les variations de l'application multilocataire sous forme d'un modèle de variabilité abstrait et uniforme. Nous nous sommes basés sur des techniques de modélisations bien connues dans la discipline de l'ingénierie des lignes de produits logiciels [Bayer& 06] [Pohl& 05].

3.8. Positionnement et bilan

Nos travaux ont eu pour principale contribution la spécification d'un nouveau méta-modèle de variabilité spécialement conçu pour modéliser la variabilité des applications SaaS mutualisées, indépendamment de leurs technologies de développement. Il s'inspire en grande partie des travaux existants sur la variabilité dans le domaine de l'ingénierie de lignes de produits logiciels. Ce méta-modèle introduit des nouveaux concepts de modélisation liés principalement aux caractéristiques émergentes du modèle SaaS mutualisé et à notre orientation de fournir la gestion de la variabilité sous forme d'un service. Nous avons pour cela introduit la notion de *VaaS (Variability as a Service)* comme un nouveau membre de la famille des services du Cloud. La justification de cette contribution réside essentiellement dans notre orientation plus générale de gérer la mutualisation de SaaS par externalisation, celui qui nécessite tout d'abord d'externaliser la gestion de la variabilité. Le méta-modèle de variabilité vient renforcer l'architecture de VaaS à travers ses concepts de modélisation proposés.

Ma contribution : j'ai encadré scientifiquement la thèse de A. Ghaddar, malgré l'année d'interruption.

Validation : les validations pratiques ont en partie été faites sur VaaS par S. Dupont durant son stage de Master 2, que j'ai encadré.

Valorisation : ces travaux ont donné lieu à 3 publications, notamment internationales dont 1 conférence de rang A, CAISE 2012 (référence [29] du CV détaillé) et 1 conférence de rang B, (référence [41] du CV détaillé) ainsi qu'une conférence nationale INFORSID 2012 (référence [78] du CV détaillé).

4. Capitalisation de migrations vers des architectures micro-services

4.1. Fiche d'identité

Titre :	Recommandations techniques et fonctionnelles pour la migration de monolithes vers des architectures micro-services
Paradigme(s) :	Architectures micro-services
Principales contributions :	. Formalisation de retours sur expérience de migration de monolithes vers des architectures micro-services. . Proposition de recommandations techniques et fonctionnelles pour réussir de telles migrations.
Mots-clés :	Micro-services ; Migration; Web Oriented Architecture, alignement métier ; reutilization par integration; retour d'expérience.
Collaborateur(s) :	J-P. Gouigoux, directeur technique MGDIS, Vannes
Mon implication :	Scientifique
Cadre de collaboration :	Aucun cadre formel.
Période :	Depuis 2017
Principales publications :	ICSA Software Architecture in Practice (SAIP) Track 2019, IEEE International Conference on Software Architecture Workshops (ICSAW) 2017.

Figure 51 : Fiche synthétique.

4.2. Synthèse

Nos travaux s'intéressent à la problématique de migration d'applications patrimoniales monolithiques vers des architectures micro-services. Nous nous basons sur l'expérience de J-P. Gouigoux, directeur technique MGDIS, un éditeur de logiciels pour le secteur public, et sur mon expertise autour des architectures logicielles et de leur évolution. MGDIS a eu à migrer leur propre monolithe en 2013 sous peine de faire faillite. La migration a duré 3 ans en ciblant une réécriture complète à partir de zéro en utilisant une architecture web moderne[18] à base de micro-services. Les objectifs initiaux étaient (i) de fournir une interface Web hautement ergonomique, (ii) d'assurer une évolution à long terme et (iii) de faciliter l'interopérabilité à faible coût avec de nombreux autres produits logiciels. Lors d'échanges à différentes occasions, nous avons décidé de tirer des leçons et d'analyser la migration effectuée afin d'en ressortir des recommandations et des bonnes pratiques que nous souhaitions faire valider scientifiquement.

La principale stratégie utilisée pour atteindre ces objectifs de migration a été basée sur l'alignement des systèmes d'information dès le début [Hirschheim& 01] [Chan& 97] [Gallier& 14] [Cassidy 16]. A savoir, les composants obtenus doivent être aussi autonomes et découplés que possible, afin de faciliter le développement et d'éviter l'encombrement éventuel de la dette technique. L'objectif était d'arriver à une application en mode SaaS [Armbrust& 10] en ciblant le style architectural des micro-services [Fowler& 14] [Newman 15].

Nos travaux ont proposé des recommandations de deux sortes :

[18] https://www.w3.org/TR/2004/NOTE-ws-arch-20040211

1. *Des recommandations d'ordre technique :* elles concernent trois préoccupations majeures et récurrentes dans les problématiques de migration de monolithes vers des architectures micro-services : déterminer la granularité des services, leur déploiement et leur orchestration. Les principales recommandations et constats ont été :
   a) *Granularité des services :* le choix de la granularité devrait être dicté par l'équilibre entre les coûts de l'assurance qualité et le coût du déploiement.
   b) *Leur déploiement :* la principale conclusion est qu'il existe un lien étroit entre les architectures basées sur les services et la technologie des conteneurs.
   c) *Leur orchestration :* un premier constat est que l'usage d'un ESB est surdimensionné, sauf pour les très grandes organisations. Tout en continuant d'utiliser un ESB, le deuxième constat est que seuls certains flux d'API particulièrement importants devraient être pris en compte pour circuler dans l'ESB. Et le dernier constat est que l'usage de moteurs BPMN risque de réintroduire des couplages.
2. *Des recommandations d'ordre fonctionnel :* nous montrons l'importance de :
   a) *Suivre une segmentation fonctionnelle :* plutôt que technique lors d'un découpage d'un monolithe pour cibler une architecture à micro-services. L'approche fonctionnelle doit guider l'approche technique.
   b) *Adopter une approche de standards/normalisation :* lorsqu'une norme existe, elle devrait être obligatoirement respectée. En l'absence de norme, une entité canonique doit être mise en place et des formats pivots doivent être mis à disposition afin d'aider à la normalisation.
   c) *A propos de la granularité de micro-services :* baser le découpage des micro-services sur des découpages fonctionnels.
   d) *Importance de la sémantique :* la sémantique s'est révélée être un défi majeur. Fixer clairement la sémantique a permis une compréhension précise et partagée des enjeux de l'entreprise.

4.3. Verrous scientifiques

L'approche micro-service a émergé de préoccupations du terrain, du monde réel. Elle a tenté de prendre en compte l'expérience et le recul issus de l'usage des SOA. Les entreprises ont beaucoup de difficultés à adopter de nouveaux styles architecturaux tels que les micro-service. Elles n'ont souvent aucune expérience et l'exercice est hautement risqué. Nous nous sommes basés sur l'expérience de migration réussie de MGDIS pour proposer des recommandations et des bonnes pratiques. Les principaux verrous adressés sont de deux ordres :

1. *Techniques :* les entreprises ne savent pas comment déterminer la granularité des micro-services, comment les déployer une fois qu'ils sont définis et comment les orchestrer. Ainsi, la principale difficulté dans la recherche de la granularité adéquate est qu'à notre connaissance, il n'existe pas d'état de l'art sur ce sujet de la granularité des services. Premièrement, il n'existe actuellement aucune définition communément acceptée de la taille souhaitée d'un micro-service [Newman 15] [Fowler& 14] [Namiot& 14]. L'unité de mesure ne connaît pas de consensus et la taille du service peut être mesurée en lignes de code, en nombre de fonctionnalités, en termes de consommation de ressources serveur dans des conditions opérationnelles standard, etc. De plus, la tendance aux micro-services s'accélère et les architectures nouvelles tendent à diminuer la taille des services, avec ce que l'on appelle les « nanoservices » (cités dans Microsoft Azure Functions et les architectures lambda [Marz& 15]/ serverless [Diot& 99]).
2. *Fonctionnels :* les entreprises font souvent l'erreur de privilégier une approche de migration par les considérations techniques, au détriment d'une approche de découpage fonctionnel. En effet,

la plupart des modèles de schémas dans l'architecture logicielle sont abordées et conçues avec des préoccupations techniques.

Nos contributions sont présentées ci-après :

4.4. Migration de monolithes vers les micro-services : recommandations techniques

Nos travaux proposent des recommandations au travers de trois questions récurrentes dans les problématiques de migration de monolithes vers des architectures micro-services : comment diviser l'ancien monolithe en API granulaires ? Comment déployer les services ? Comment faire fonctionner les services ensemble ?

a. Granularité des services

Comment diviser l'ancien monolithe en API granulaires ? L'expertise technique n'aide en rien. Beaucoup de littérature explique l'approche micro-services, mais elle est le plus souvent citée dans le contexte d'applications web à très haut volume, ce qui n'était pas le cas du MGDIS. Le premier retour d'expérience que nous proposons est que le choix de la granularité devrait être déterminé par l'équilibre entre les coûts de l'assurance qualité et le coût du déploiement, comme le montre la Figure 52.

Le coût de l'assurance qualité (QA) peut être calculé sur une version donnée en additionnant le temps passé par les testeurs à valider non seulement les nouvelles fonctionnalités mais aussi la non-régression sur les fonctionnalités existantes avec le temps passé sur la gestion des versions. Le coût du déploiement est le temps passé par les équipes opérationnelles pour déployer cette nouvelle version, également en jours-homme. Elle diminue significativement à mesure que les équipes automatisent cette opération.

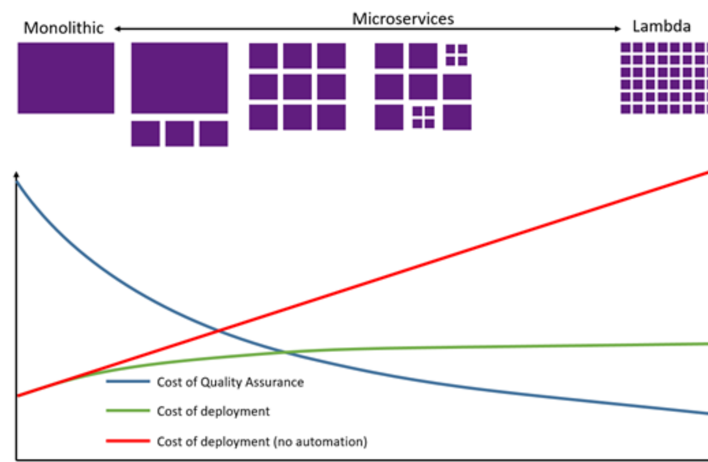


Figure 52 : Détermination de la granularité des services en fonction des coûts.

Le temps de déploiement ne dépend généralement pas du nombre de fonctionnalités supplémentaires de la version. Quant à l'assurance qualité, l'automatisation des tests de non-régression aide à maintenir un faible coût pour les fonctionnalités existantes, mais les nouvelles sont accompagnées de tests manuels coûteux ou de la rédaction de scénarios de tests automatisés. Ainsi, la comparaison des deux coûts doit être liée à la valeur ajoutée métier des nouvelles fonctionnalités, généralement estimée

avec des méthodes agiles en jours/homme ou en points de fonctionnalités si l'on veut être indépendant de la vitesse de l'équipe.

Sur le côté droit de la Figure 52 se trouve l'architecture à grain fin, à savoir l'architecture lambda. Sur le côté gauche, se trouve le service le plus grossier possible, à savoir l'architecture monolithique où un seul processus logiciel intègre toutes les fonctions exposées. Dans l'architecture logicielle monolithique, le couplage entre les composants est massif. Il en résulte un coût énorme pour l'assurance qualité, car une modification peut avoir un impact sur n'importe quelle caractéristique de l'application. Le coût d'exploitation d'une telle application est toutefois limité, puisqu'il n'y a qu'un seul processus pour l'installer, la surveiller et la maintenir en bon état. Au fur et à mesure que l'on progresse vers la droite de la Figure 52, la granularité s'affine, ce qui a deux conséquences. Premièrement, le coût de l'assurance qualité (ligne bleue) diminue, puisqu'il devient possible de tester et de valider les services un par un, puisqu'ils sont réellement indépendants (au moins du point de vue des tests unitaires et des tests fonctionnels). Deuxièmement, le coût de fonctionnement de l'ensemble augmente avec le nombre de services (lignes rouge et verte). Le point de franchissement de la ligne bleue et rouge indique que l'architecture est optimale. Il est à noter que les coûts diffèrent selon que le déploiement soit automatisé ou manuel.

Cette tentative de modélisation de la granularité des services a été validée dans la pratique par le fait qu'elle correspondait bien à d'autres critères de granularité, par exemple l'alignement du service sur les normes et standards de l'entreprise, ou l'utilisation de la notion de Contexte Borné au sens défini par Evans dans Domain-Driven Design [Evans 03].

b. Déploiement des services

Comment déployer les services ? lorsque les services ont été identifiés et définis, comment leur déploiement peut-il être réalisé, puisque les méthodes traditionnellement utilisées pour les applications monolithiques ne sont plus adaptées ?

La principale conclusion est qu'il existe un lien étroit entre les architectures basées sur les services et la technologie des conteneurs. Les technologies basées sur les conteneurs, et en particulier son implémentation la plus connue, Docker, offrent des caractéristiques intéressantes :

- Une image Docker contient toutes ses dépendances, ce qui signifie qu'un service donné peut être traité comme une boîte noire, exposant seulement son API en échange de ressources.
- Les conteneurs sont par défaut scellés l'un à l'autre, ce qui permet de garantir un faible couplage, sans le coût élevé associé aux machines virtuelles.

c. Intégration de services

Comment faire fonctionner les services ensemble une fois qu'ils sont définis et installés ? Dans la littérature et dans les faits, au fur et à mesure que l'architecture SOA était remplacée par une architecture orientée Web, les bus de services d'entreprise s'estompèrent pour laisser place à des techniques d'intégration à faible encombrement, telles que les communications asynchrones basées sur RSS [Shah& 10] [Bieberstein& 08] ou les événements basés sur Webhooks [Jaramillo& 16] ainsi que les échanges de données entre services.

Après plusieurs années d'essais de ces différentes approches, il ressort que le point fort est l'intégration des micro-services avec des webhooks. Ils représentent un moyen de communication léger, discret et basé sur HTTP entre APIs. Les ESB sont laissés à l'initiative des grands clients lorsqu'ils veulent se charger de connecter les événements Webhook aux API d'une manière différente

et plus sophistiquée que de simplement connecter la sortie de l'événement à l'API sans aucune médiation. Ainsi, le comportement par défaut est la chorégraphie passive, et le client est capable d'insérer un middleware afin de passer à l'orchestration active.

d. Bilan

L'expérience montre que les projets SOA ont souvent été décevants, contrairement à ce qui a été affirmé, en raison de la faible réutilisation réelle des services. La plupart des applications réellement orientées services n'ont pas obtenu les avantages promis de la SOA [Hirschheim& 01] [Chan& 97] [Galier& 14]. En migrant vers une architecture orientée web et en nous appuyant sur des micro-services appropriés, nous avons extrait trois principaux bénéfices :

1. *Augmentation de la réutilisation :* la plupart des services fraichement prêts pour la production ont pu être réutilisés rapidement.
2. *Remplacement facilité :* la réutilisation est certainement la caractéristique la plus recherchée d'un service, car elle entraîne une réduction automatique des coûts, mais la facilité de remplacement est un effet secondaire positif des services bien construits.
3. *Augmentation du rendement :* après la réutilisation et le remplacement, la performance est le troisième grand avantage observé en production par MGDIS sur sa nouvelle architecture.

4.5. Migration de monolithes vers les micro-services : recommandations fonctionnelles

Ces travaux sont une continuité des précédents. Ils visent à les généraliser vers de bonnes pratiques issues du processus de migration. Ces bonnes pratiques que nous proposons sont :

a. Suivre une segmentation fonctionnelle

La question revient : comment choisir de découper l'ancien monolithe en API granulaires ? La principale recommandation est que l'approche fonctionnelle doit guider l'approche technique. L'utilisation de la représentation en quatre couches (Figure 53) du livre blanc du CIGREF [CIGREF 03] a considérablement contribué à la réussite du projet en offrant un moyen approprié d'associer le fonctionnel au technique. Un retour d'expérience essentiel de la migration industrielle est que les plans de coupe ne doivent pas provenir de la couche 3 (technique) mais de la couche 2 (fonctionnelle), car celle-ci englobe toutes les fonctions métier qui sont ensuite techniquement mises en œuvre au niveau 3.

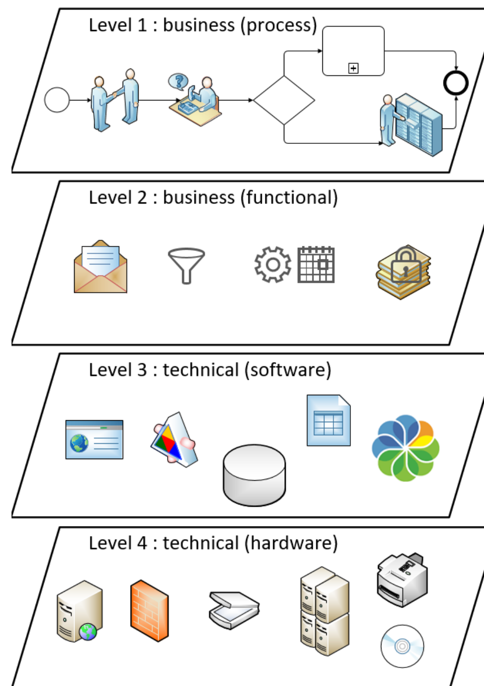


Figure 53 : Diagramme à quatre niveaux.

b. *Adopter une approche de standards/normalisation :*

Lorsqu'une norme existe, elle devrait être obligatoirement respectée. En l'absence de norme, une entité canonique doit être mise en place et des formats pivots doivent être mis à disposition afin d'aider à la normalisation. L'utilisation de normes est une exigence bien acceptée dans l'architecture d'entreprise, mais pas toujours respectée. Les micro-services dédiés à la prise en charge d'une norme métier permettent un bon alignement entre les logiciels et les processus [Edwards& 01]. Ils représentent en fait la « colle » entre la couche fonctionnelle et la couche technique dans le diagramme à quatre couches. L'utilisation d'un point de vue fonctionnel plutôt que technique pour découper les services encourage à utiliser normes et standards ainsi qu'une approche *Domain Driven Design* [Evans 03] pour établir un format pivot interne et approprié et des entités canoniques en l'absence de normes. Ces normes, standards, format pivot et entités canoniques sont utilisés pour la définition des contrats de micro-services. Leur utilisation pour définir l'architecture s'est avérée être le changement majeur en termes d'impact sur l'application résultante.

Dans l'exemple de MGDIS, l'application finale utilise des dizaines de normes métier (vCard, OAuth2, SAML, SCIM, XACML, GeoJSON, CMIS, ODPv2 pour n'en citer que quelques-unes) et des centaines de normes techniques (RFC, IETF, ISO, etc.). Il est intéressant de noter que les normes et standards proviennent généralement d'un consortium d'experts du domaine métier qui s'entendent sur une représentation précise (donc technique) de leur domaine, à l'image de cette initiative *IT Standards for Business*[19].

[19] https://www.itforbusiness.org/

c. A propos de la granularité de micro-services :

L'utilisation du préfixe « micro » tend à exprimer le fait que la taille réelle du service est importante. Pourtant, les avis sont multiples sur la taille d'un micro-service [Newman 15] [Fowler& 14] [Namiot& 14]. Il n'y a même pas un accord sur l'unité qui devrait être utilisée (lignes de code, poids de la bibliothèque, nombre de méthodes dans l'API, etc.). Le domaine est également flou avec l'apparition des nanoservices et des architectures lambda [Marz& 15]/ serverless [Diot& 99]).

Notre principale recommandation est d'abandonner ces considérations techniques et de baser la découpe des micro-services sur des considérations fonctionnelles, comme pour n'importe quel composant correctement réalisé. Au lieu d'utiliser les notions du niveau 3, cela signifie utiliser la sémantique du niveau 2 du diagramme (Figure 53).

d. Importance de la sémantique

La sémantique s'est révélée être un défi majeur. Fixer clairement la sémantique a permis une compréhension précise et partagée des enjeux de l'entreprise. Comme tous les micro-services doivent être facilement remplaçables, la sémantique de l'entreprise a progressivement pris une place de plus en plus importante. Ceci est étroitement lié à l'approche DDD [Evans 03] qui doit concilier les utilisateurs commerciaux et techniques. La sémantique est la base de la deuxième couche du diagramme. Le fait de créer les formats pivots et les entités canoniques, avec les vocabulaires et définitions normalisés associés, a permis une compréhension précise et partagée des enjeux de l'entreprise. Raisonner en termes de sémantique et choisir le mot juste pour chaque situation a aidé à résoudre des situations de plus en plus complexes à l'aide de logiciels. Les expressions, et en particulier le jargon commercial, sont souvent des formulations trop simplifiées d'une situation plus complexe, et l'amélioration de la sémantique a beaucoup aidé à trouver le bon modèle pour un concept commercial.

4.6. Contexte des travaux (thèses, M2, contrat, collaborations)

Collaboration depuis 2017 sans cadre formel avec Jean-Philippe Gouigoux, directeur technique de l'éditeur logiciel MGDIS.

4.7. Brève synthèse des travaux fondateurs

Il n'y a pas, à notre connaissance, de travaux fondateurs sur le retour d'expérience quant aux migrations de monolithes dans les entreprises vers les micro-services. Les besoins sont récents. Les travaux se focalisent généralement sur un point de vue précis pour mener leur migration. Ainsi, nous pouvons citer [Balalaie& 16] qui présentent les leçons tirées de leur migration mais qui nous paraissent assez triviales telles que le déploiement dans l'environnement de développement est difficile, le développement de systèmes distribués a besoin de développeurs qualifiés ou encore les micro-services ne sont pas une solution miracle. De plus, les considérations sont essentiellement techniques. Les travaux de [Francesco& 18] représentent une enquête menée auprès de développeurs ayant fait des migrations afin d'étudier les différentes activités misent en place pour y arriver. Ils fournissent des études statistiques pour tenter d'identifier des idées plus généralisables et communes aux acteurs industriels. Nos travaux font partie de cette enquête.

4.8. Positionnement et bilan

Les principales contributions de nos travaux sont l'analyse et la formalisation de retours d'expérience industrielle sur la migration de logiciels monolithes vers des architectures micro-services sur les

aspects techniques et sur les aspects fonctionnels. Ces travaux s'appuient sur une migration d'un monolithe menée sur 3 ans et dégagent deux catégories de leçons :

1. Les leçons techniques apprises pour une migration réussie et adaptée au contexte vers une architecture micro-services sont des recommandations pour l'identification de la granularité des services, leur déploiement et leur orchestration.
2. Les leçons d'un point de vue fonctionnelles ont montré l'importance d'avoir une approche fonctionnelle, des normes et standards, de la granularité des micro-services et leur sémantique, et enfin les résultats techniques et d'intégration.

Ma contribution : identifier les aspects clés à ressortir comme des recommandations et des bonnes pratiques à des fins de validation scientifiques.

Validation : les travaux sont des retours d'expérience.

Valorisation : ces travaux ont donné lieu à 2 publications internationales : ICSA Software Architecture in Practice (SAIP) Track 2019 (référence [67] du CV détaillé) et 1 workshop AWS de IEEE International Conference on Software Architecture Workshops (ICSAW) 2017 (référence [69] du CV détaillé). Notre collaboration est toujours en cours.

5. Conclusion

Ces travaux menés avec mes collaborateurs sur les architectures logicielles autour de la notion de service se poursuivent car les problématiques abordées sont toujours d'actualité. Les travaux liés au SaaS mutualisé ne sont qu'un début et les défis à relever sont encore importants et cruciaux. Les problématiques liés aux systèmes d'information d'entreprises, notamment les workflows inter-organisationnels restent d'une sensibilité accrue avec le décloisonnement des entreprises lié aux nouvelles tendances du numérique (IoT, Big Data, Cloud, Fog, Edge computing et même l'Intelligence Artificielle). Les travaux liés au micro-services dans le cadre de l'alignement Businee-IT ne sont qu'au préambule. Il est à noter que l'expérience tirée des deux catégories de travaux alimentent une nouvelle voie que j'explore : la spécification d'une architecture logicielle générique du Jumeau Numérique dans le cadre de l'Industrie 4.0.

Imprégnée de cette conclusion, de ces différents travaux, des échanges au gré de rencontres enrichissantes ainsi que des échecs rencontrés, je dresse un bilan et détaille ci-après mes perspectives scientifiques :

Chapitre 5.

Conclusion et Perspectives

SOMMAIRE

2. Bilan ____ 115
2.1. Contributions ____ 115
a. Sur les architectures logicielles à base de composants (2002 – 2010) ____ 115
b. Sur les architectures logicielles autour du service (depuis 2010) ____ 116
2.2. Echecs ____ 116
3. Perspectives scientifiques : vers une industrialisation du Génie Logiciel ____ 117
3.1. Introduction ____ 117
3.2. Industrialiser le Génie Logiciel ? ____ 117
3.3. Projets scientifiques ____ 118
a. Vers une adoption aisée du SaaS mutualisé ____ 118
b. Architecture logicielle générique pour le Jumeau Numérique ____ 119
c. Vers la prise en compte de la stratégie d'entreprise ____ 121
d. Architecture logicielles continues et aide à la décision dans un environnement complexe et incertain ____ 123
e. Collaborations scientifiques ____ 123
4. Perspectives vers la formation universitaire et le monde socio-économique ____ 123
4.1. Valorisation de la formation universitaire par la recherche ____ 123
4.2. Valorisation des collaborations avec le monde socio-économique ____ 124

1. Introduction

Pourquoi est-il primordial de pouvoir faire évoluer un système logiciel ? Pour lui assurer une durabilité dans le temps tout en préservant idéalement de bonnes qualités et éviter des coûts trop importants. Pourquoi la maintenance et l'évolution logicielles sont-elles si complexes et si coûteuses ? Il existe bien entendu différentes raisons. Une des raisons primordiales à mon sens, est le

peu de séparation de préoccupations. Une autre raison que j'identifie est le manque de normes et standards pour le logiciel. « *Automobile, transport aérien, énergie électrique.... Chaque fois qu'un secteur d'activité est passé à la dimension industrielle, les clients en ont tiré de très nombreux avantages en termes de coûts, de fiabilité et de choix* », Louis Naugès[20]. Dans toutes les industries, des *composants standards* et *interchangeables* sont utilisés, à l'exemple des pneus ou des essuie-glaces pour l'industrie automobile.

De nos jours, le seul domaine du numérique, à ma connaissance, qui a réussi à mettre en place des standards qui sont incontournables dans un usage quotidien sont autour du web et ses différents usages. Ainsi, les langages du web (HTML, XHTML, XML, CSS...) reposent sur des normes et des standards. Les organisations de normalisation, comme le W3C (World Wide Web Consortium), créent un consensus à travers ces groupes et ces experts pour maintenir et développer des principes architecturaux cohérents. Utiliser une norme permet la compatibilité du langage et des interactions mais aussi une plus grande pérennité. Utiliser une norme assure une ouverture au niveau mondial. A mon sens, cet aspect gagnerait à être renforcé pour le Génie Logiciel. C'est ce que je qualifie d'*industrialisation du Génie Logiciel*.

Il est primordial d'arriver à rationaliser et optimiser toutes les infrastructures numériques à proprement parlé tout autant que les organisations sous-jacentes. Il faut optimiser les ressources, leur utilisation tout en assurant des niveaux élevés de qualité de services et tout en réduisant les coûts et les risques. Je suis convaincue que pour arriver à *industrialiser le génie logiciel* il faut :

1. Une séparation stricte de toutes les préoccupations logicielles.
2. Un développement logiciel et par conséquent une maintenance et une évolution basé sur des standards et des normes, ou à défaut sur des référentiels par domaine qui font consensus.

Cela passe par intégrer non plus uniquement les architectes logiciels dans le processus mais également tout l'environnement économique impactant les décisions (infrastructures, usages industriels, fournisseurs, DSI) et surtout définir le rôle des dirigeants dans ce changement.

2. Bilan

Dans mes travaux de recherche, j'ai toujours suivi une séparation stricte des préoccupations, d'abord de manière inconsciente dans mes jeunes années et ensuite avec une conviction croissante. J'apporte en cette fin de document un éclairage probablement partial et contextuel :

2.1. Contributions

Une des contributions dont je suis la plus heureuse est d'avoir participé, par l'encadrement, au travail de thèse d'une dizaine de doctorants dont une bonne partie est dans la vie active soit en tant qu'enseignant-chercheur soit en tant que cadre en entreprise. Dans le cadre de leurs travaux ainsi que dans le cadre de collaborations avec des confrères, nous avons pu apporter un certain nombre de contributions :

a. Sur les architectures logicielles à base de composants (2002 – 2010)

La première contribution, dans le cadre des travaux de thèse de Nassima Sadou, a été un modèle générique pour l'évolution structurelle d'architectures logicielles. Ces travaux ont permis de réfléchir

[20] R2I : la Révolution Industrielle Informatique - Première partie.

à comment spécifier l'évolutiosn tout en la dissociant des systèmes sur lesquelles elle portait. La deuxième contribution, dans le cadre des travaux de thèse de Olivier Le Goaer, a été de capitaliser et formaliser un savoir-faire d'évolution, souvent non formalisée, et de l'encapsuler au sein de *styles d'évolution*. Cette contribution nous a valu plusieurs citations de David Garlan, un des pères fondateurs de la discipline, et de son équipe de Carnegie Mellon University. Le hasard a fait que nous avions proposé en parallèle le concept de style d'évolution mais avec des sémantiques et des objectifs différents. La troisième contribution, dans le cadre des travaux de thèse de Sylvain Chardigny, a été de proposé une approche d'extraction d'une architecture logicielle représentative d'un code objet existant. Cette approche quasi-automatisée a l'avantage non seulement de prendre en compte l'analyse d'informations factuelles tel que le code mais également de la documentation et de l'architecture intentionnelle. Une des publications reste la plus citée parmi la liste de mes publications. Et, enfin, la dernière contribution présentée, dans le cadre d'une collaboration avec Tom Mens de l'Université de Mons en Belgique, a été de s'appuyer sur la théorie des graphes pour proposer des patrons d'évolution architecturale. Elle représente ma première collaboration internationale.

b. Sur les architectures logicielles autour du service (depuis 2010)

Le changement de paradigme des composants vers les services s'est en bonne partie fait dans le cadre de collaborations avec des entreprises. Ces travaux ont confirmé mon intérêt de travailler avec le monde socio-économique. L'idée est d'ouvrer à mettre en place des leviers économiques.

La première contribution, dans le cadre des travaux de la thèse CIFRE de Ali Ghaddar, a été un modèle d'externalisation de la gestion de la variabilité d'applications SaaS mutualisées. La variabilité et sa gestion sont alors proposées comme un service à part entière mis à disposition de fournisseurs d'applications SaaS mutualisées. L'atout majeur est qu'il permet de les décharger des préoccupations complexes liées au SaaS mutualisé. La deuxième contribution, dans le cadre des travaux de thèse de Souad Boukhedouma, a été de proposer une approche complète pour introduire de la flexibilité dans les workflows inter-organisationnels des entreprises. L'approche s'est basée sur les principes SOA pour tirer profit notamment de sa flexibilité. Plusieurs patrons de coopération ont été proposés ainsi que deux patrons d'évolution et d'adaptation pour ces workflows. La dernière contribution présentée, dans le cadre de collaboration avec Jean-Philippe Gouigoux, directeur technique de l'éditeur logiciel MGDIS, a été de faire ressortir des recommandations et de bonnes pratiques pour migrer avec succès des systèmes monolithes vers des architectures micro-services. Cette collaboration n'a pas de cadre formel. Elle est entièrement basée sur notre volonté de faire converger préoccupations économiques et préoccupations scientifiques. Elle se poursuit actuellement sur des anti-patterns d'alignement ainsi que la mise en place d'une collaboration avec Joost Noppen, principal researcher à British Telecom.

2.2. Echecs

Parmi mes regrets et qui restent une des préoccupations qui me taraudent, figure l'incapacité de valider à grande échelle des résultats théoriquement validées. Les travaux menés sont essentiellement conceptuels, avec des validations à petite échelle. Les solutions proposées prennent du temps, temps particulièrement compté durant les thèses. Ce manque de validation pratique à large échelle a plusieurs fois entravé de peu des publications dans des journaux de rang A et A*. Cela m'amène à réfléchir différemment mon organisation et notamment mes futurs encadrements. Cela m'amène également à concevoir mes projets de recherche différemment avec des objectifs conceptuels plus fins et des collaborations avec des expertises complémentaires, notamment au sein de mon équipe NaoMod.

Mon second regret concerne des travaux réellement prometteurs qui n'ont pas pu se poursuivre par manque de moyens, humains surtout et financiers. Je prends pour exemple les travaux issus de la thèse de Ali Ghaddar sur l'externalisation de la gestion de la variabilité des applications SaaS mutualisé. Les pistes ouvertes sont aussi prometteuses sur le plan scientifique qu'économique. Cela m'amène à poursuivre les échanges avec le monde socio-économique pour le montage de projets collaboratifs.

3. Perspectives scientifiques : vers une industrialisation du Génie Logiciel

3.1. Introduction

Initialement, la discipline des architectures logicielles avait une approche descriptive, concevant une abstraction réutilisable (un modèle) d'un système logiciel. Cette approche a permis de raisonner et d'agir sur le système dans son ensemble, amenant à construire des structures plus élaborées à différents niveaux de conception du logiciel. Par la suite, la discipline a évolué en faveur d'une approche prescriptive, mettant l'accent sur l'importance des décisions en matière de conception architecturale et leur incidence sur les qualités du système, ainsi que le rôle des connaissances architecturales en tant que base de l'architecture commune. Les deux approches se sont établies et sont devenus un standard dans le développement de logiciels.

L'avènement de la troisième génération d'architectures logicielles et ses nouveaux paradigmes permettent le développement de nouvelles technologies, notamment pour les villes intelligentes et l'industrie du futur. Cependant, les nouvelles technologies soulèvent de nouveaux défis, y compris l'évolutivité. Elles dessinent l'économie du futur et doivent par conséquent pouvoir faire face aux défis associés. Elles nécessitent de nouvelles solutions architecturales facent auxquelles les expertises actuelles sont insuffisantes. La montée en puissance du Big Data, des systèmes hybrides ou des systèmes cyber-physiques lève de nouvelles préoccupations et impacte grandement la complexité, la réactivité et le passage à l'échelle. Les différents modèles du Cloud computing, les conteneurs, les micro-services, architectures basées sur les événements, la virtualisation, les SDN (software-defined network), l'ubiquité et l'IoT élargissent le spectre des préoccupations, ajoutant celles liées à la décentralisation. A ces différentes préoccupations, s'ajoutent l'intégration des concepts de l'Intelligence Artificielle pour poser le cadre de la future génération d'architectures logicielles. Ce contexte de nouvelles architectures ouvre des enjeux cruciaux d'intégration, de réactivité et d'adaptation. Les enjeux et risques d'évolution sont décuplés par rapport aux systèmes logiciels classiques.

3.2. Industrialiser le Génie Logiciel ?

L'industrialisation des processus liés aux infrastructures est l'ensemble des actions entreprises pour rationnaliser, standardiser, optimiser les services rendus par les infrastructure informatiques, internes ou externes, de l'organisation. L'industrialisation permet d'optimiser l'utilisation des ressources, d'assurer des niveaux de qualité de service, de réduire les coûts et les risques. Ainsi, même si les améliorations et les progrès substantiels dans le domaine de l'architecture logicielle ont permis au logiciel d'atteindre une complexité sans précédent, il reste encore beaucoup de travail pour réaliser la vision de l'architecture logicielle en tant qu'une discipline d'ingénierie à part entière. L'intérêt renouvelé du monde socio-économique pour les nouveaux domaines du numérique montre que les

attentes sont importantes. Mon projet scientifique s'intéresse ainsi à la prochaine génération des architectures logicielles.

Les logiciels, pour la plupart, sont conçus et opèrent en « circuit fermé ». Au-delà du principe orienté service, mon idée principale est de tirer profit du terrain très propice du modèle SPI (IaaS, SaaS et PaaS) du Cloud et de la maturité de ses offres et d'y coupler des solutions hybrides (externes et internes). En effet, les solutions SaaS permettent de concevoir des logiciels industriels, fiables et robustes. Les solutions SaaS, surtout les solutions SaaS multitenants, peuvent être d'excellentes réponses.

Ainsi, le principal enjeu qui m'anime pour les années à venir et d'œuvrer vers une industrialisation du logiciel au travers des futures générations d'architectures logicielles. L'approche que j'envisage reste une approche par et pour la réutilisation et en ciblant des évolutions semi-automatisées sous forme d'assistance, avec un intérêt particulier pour l'évolution en continue.

3.3. Projets scientifiques

Les trois projets présentés ont des bases déjà posées :

a. Vers une adoption aisée du SaaS mutualisé

Ce projet vient en continuité des travaux de thèse de A. Ghaddar (Chapitre 4. 3.) dédiés à l'externalisation de la gestion de la variabilité et à l'isolation des données des applications développées selon le mode SaaS mutualisé du Cloud. L'objectif est de regrouper ces premières contributions ainsi que d'autres services liés à l'administration et à la sécurité des applications mutualisées dans une plateforme dédiée, vers une approche globale de gestion de ce type d'applications par externalisation.

Le projet vise ainsi une solution de migration non intrusive et faiblement couplée vers le style architectural SaaS mutualisé. Le SaaS mutualisé réduit les CAPEX-OPEX grâce à la mutualisation et au partage de ressources dans le Cloud. Or les entreprises y sont réticentes au vu des compétences nécessaires, du changement technologique et de la forte dépendance aux plateformes SaaS. Le projet vise une migration faiblement intrusive vers le SaaS mutualisé et un hébergement libre dans le Cloud, grâce à un processus de migration et une plateforme tous deux faiblement couplés. Ce projet nécessite des expertises académiques sur l'évolution d'architectures Cloud, l'intégration de plateformes Cloud et le test de modèles. Il sera également indispensable d'intégrer des entreprises. Sur le plan stratégique, le projet propose de préserver l'environnement technologique des organisations, l'intégrité des systèmes d'information (SI) et les applications découplées techniquement des fournisseurs de Cloud. Conceptuellement, il proposera une plate-forme fournissant un ensemble de composants de base facilement extensibles, s'intégrant à un processus de migration. Sur le plan technologique, la plate-forme sera une plate-forme PaaS (Platform-as-a-Service) fournissant des applications à faiblement couplé au niveau SaaS. Il sera proposé sous la forme d'un « environnement groupé » sur des infrastructures virtuelles.

Ce projet a déjà été murement réfléchi. Il a déjà été déposé comme projet PRCE à l'ANR en 2017. Il n'a pas été sélectionné en raison d'une faible présence indsutrielle, mais les retours étaient encourageants. La problématique économique est toujours d'actualité. Parmi les problématiques abordées, celles en lien avec les expertises internes à mon équipe NaoMod. A noter celle des tests sur les résultats de migration dont l'approche envisagée est proposée par Jean-Marie Mottu. Ce projet

gagnerait également à travailler en étroite collaboration sur le projet Lowcomote21 qui cible un environnement de développement intégré (IDE) visuel dans lequel des développeurs citoyens peuvent ajouter des composants d'application par glisser-déposer et les connecter entre eux pour créer des applications. *Low-code development platforms* est un projet Européen qui finance 15 thèses au niveau de l'Europe. Ce projet est porté par mon équipe NaoMod et dont Massimo Tisi est le porteur.

Les Lowcode Engineering Platforms (LCEPs) sont ouvertes, permettant d'intégrer des outils d'ingénierie hétérogènes, interopérables, permettant une ingénierie multiplateforme, évolutive. Lowcomote aurait un apport très pertinent avec l'objectif du projet, notamment sur la partie externalisation de la gestion du SaaS mutualisé et les interactions avec les experts métier. Cette piste d'une Lowcode Engineering Platforms pour le développement d'une plate-forme de migration faiblement couplée d'une application vers du SaaS mutualisé est envisageable à court terme.

Par ailleurs, des échanges sont en cours en vue d'une collaboration avec Mohammad Abu Matar Regis University, Information Technology Department, Denver, USA, sur les plateformes d'évolution dynamiques d'applications SaaS mutualisé.

b. Architecture logicielle générique pour le Jumeau Numérique

Le Digital Twin permet aux industries d'intégrer les technologies clés du numérique : celles autour du logiciel et de son exécution (génie logiciel, architecture logicielles, Cloud, IoT) et celles autour de l'exploitation efficace et efficiente de la donnée (Big Data, IA…). Les grands groupes ont déjà pris le virage, les industries de tailles moyenne à petite sont plus en peine à le faire. Toute industrie gagnerait à adopter ces nouvelles technologies avec pour objectifs principaux viabilité, compétitivité et croissance économiques. Le concept du Digital Twin a été introduit pour la première fois par Michael Grieves en 2002 à l'Université du Michigan. A cette époque, les industriels travaillent avec des produits physiques ou virtuels puisque les technologies utilisées les désactivaient pour créer la connexion bidirectionnelle entre les mondes physique et virtuel. Les modèles virtuels étaient une mauvaise représentation des produits physiques puisque les données recueillies à partir des produits physiques étaient limitées, parfois indisponibles pendant le cycle de vie du produit et ne pouvaient être collectées et analysées automatiquement. Ce manque d'information et de données sur le produit physique s'ajoutait au manque de technologies pour la collecte et l'analyse des données nécessaires au développement du modèle virtuel et au manque de technologies de communication de l'information pour relier les produits physiques au modèle virtuel. La situation empêchait les industriels de travailler simultanément avec le monde physique et virtuel. Le Digital Twin a été proposé comme solution à la fusion cyber-physique.

L'importance du Digital Twin augmente avec le développement constant de nouvelles technologies qui offrent de plus en plus de possibilités, mais cela ne se fait pas sans peine. Les collaborateurs travaillant sur un même produit sont concentrés chacun sur leur partie, induisant des activités en silos et cloisonnées entre les différentes parties prenantes travaillant sur ce même produit. Il arrive que de nouvelles solutions soient développées alors que des solutions similaires existent déjà, que ce soit en interne ou non. Il arrive que cela induise, de manière totalement involontaire, des effets de bord négatifs d'autres activités de l'industrie, tels que modifier une pièce pour améliorer les performances et induire des coûts de maintenance élevés ou impacter la chaine d'approvisionnement ou de production. Il est primordial d'avoir une vision d'ensemble de toutes les activités autour d'un produit

[21] https://www.lowcomote.eu/

tout en maintenant la cohérence des différentes vues et répondant aux attentes des différentes parties-prenantes. Ce qui pousse à une réflexion approfondie sur sa conception et à offrir une architecture logicielle robuste et fiable. Le projet vise à spécifier cette architecture logicielle générique, minimale et évolutive d'un Digital Twin. Il vise également à travailler à la mise en place de connexions efficace et fiable vers et du Digital Twin malgré les contraintes rencontrées au sein des systèmes physiques de production (nœuds peu fiables, connectivité intermittente, problème de temps de latence…).

Depuis mai 2019, je suis membre du groupe de travail de l'AFNOR au sein de la Commission de Normalisation sur l'Ingénierie des Données et des Modèles pour l'Industrie (AFNOR/CP IDMI) pour la norme ISO/CD 23247 pour le Jumeau Numérique dans l'industrie (*Digital Twin manufacturing framework*). Cette série ISO 23247 démarre tout juste. Elle vise à définir un cadre pour la fabrication de Jumeau Numérique comme des représentations virtuelles d'éléments physiques de fabrication tels que le personnel, les produits, les actifs et les définitions de processus. Le *Digital Twin Manufacturing* est la modélisation détaillée des configurations physiques et la modélisation dynamique des changements de produits, de processus et de ressources au cours de la fabrication. Voici les portées des 4 parties de cette série ISO 23247. Je m'intéresse essentiellement à la partie 2 pour la définition d'une architecture de référence :

- *Partie 1 - vue d'ensemble et principes généraux :* fournit une vue d'ensemble du Digital Twin Manufacturing. Cette partie décrit les principes généraux pour donner des orientations pour le développement d'un cadre de fabrication Digital Twin ;
- *Partie 2 - Architecture de référence :* fournit une architecture de référence, un ensemble de termes et de définitions et les exigences pour la réalisation de la fabrication de Digital Twin en termes de modélisation de l'information, de simulation en boucle, d'échange d'information et d'identification des objets d'information ;
- *Partie 3 - Représentation numérique des éléments physiques de fabrication :* la cartographie des éléments de fabrication définis à l'externe, y compris les définitions des produits, des procédés et des ressources avec leurs caractéristiques, afin de fournir des représentations numériques pour la fabrication de jumelages ;
- *Partie 4 - Échange d'informations :* identifie les technologies telles que les protocoles réseau, API, langages de description, etc., pour la synchronisation, l'échange et la gestion de l'information des jumeaux de fabrication représentés numériquement.

Ce projet d'architecture logicielle générique pour le Jumeau Numérique prend son ancrage dans la thèse de Cyrine SELMA, que je co-encadre avec Olivier Cardin et Nasser Merbarki, collègues du LS2N. Mon encadrement est sur la spécification logicielle du jumeau numérique d'un système de pilotage de systèmes cyber-physiques de production appliqué aux centres de tri postaux de nouvelle génération.

Par ailleurs, je participe au Groupe de Travail Thématique « Jumeau Numérique » organisé par le pôle de compétitivité EMC2. Dans ce cadre, j'ai initié le projet d'architecture logicielle générique pour le Jumeau Numérique. Ce projet intéresse Airbus, Stelia Aerospace et Naval Group. Il intéresse également des confères du LabSTICC, Éric Martin, Joël Champeau et Sylvain Guérin. Éric Martin notamment est un spécialiste du Jumeau Numérique et l'Industrie 4.0. Nous envisageons de mettre en synergie nos compétences pour offrir un socle scientifique solide sur le sujet et couvrant tout le Grand Ouest en France. Ce projet s'intègre également dans les objectifs scientifiques de mon équipe NaoMod sur les répliques numériques d'un objet, d'un processus ou d'un système qui peut être utilisé à différentes fins.

c. Vers la prise en compte de la stratégie d'entreprise

Mes premiers travaux considéraient l'architecte logiciel comme le référant omniscient et ultime. Au fur et à mesure de mon expérience et surtout des leçons tirées de mes succès, de mes échecs et des différents échanges, l'importance d'ouvrir le champ de l'information et des décisions à des niveaux stratégiques d'une organisation s'est imposée comme une évidence, réveillant ma formation initiale en Systèmes d'Information. Le volet qui m'intéresse est l'alignement entre le métier et son support logiciel. Par l'alignement Business-IT, en considérant [Luftman 04], [Luftman& 17], [Horlach& 16], [Wu& 15], j'entends la capacité dynamique d'une entreprise à utiliser, de manière efficace et cohérente, les technologies du numérique pour atteindre ses objectifs et améliorer sa compétitivité. Pour réussir, l'alignement entre le numérique et l'entreprise doit incarner le numérique dans la stratégie comme un moyen et non comme un objectif. L'importance de l'informatique est d'une telle ampleur qu'elle est devenue économiquement vitale au travers de ses différentes Technologies de l'Information (TI) (Information Technologies (IT) en anglais). Elle s'invite dans les décisions d'orientations stratégiques dans tous les secteurs socio-économiques. Pour toutes les organisations, l'IT est passé ces 10 dernières années d'une vision technique vers une vision socio-technique, avec l'explosion de l'usage des technologies de l'information, embarquant ensemble, bon an mal an, personnes, systèmes d'information, systèmes de production (de biens et/ou services) et systèmes techniques le tout souvent empreint d'une grande complexité. Depuis, le terme numérique (digital en anglais) prend le devant de la scène, suscitant modes et débats.

Les promesses de performance et de compétitivité des stratégies numériques conduisent les entreprises et les industries à s'orienter massivement vers l'adoption et la mise en œuvre de nouvelles opportunités liées au numérique. Une telle stratégie implique de transformer les systèmes existants en processus métier modulaires, distribués, interfonctionnels et globaux. Les problèmes d'alignement entre les processus métiers et leurs systèmes logiciels sous-jacents deviennent beaucoup plus sensibles et augmentent même dans des contextes hautement dynamiques. Des travaux récents réfutent l'idée que l'alignement est juste un évènement ou encore un état final [Benbya& 06], mais plutôt un périple d'adaptations et de changements continus [Karpovsky& 15], [Wilson& 13]. Benbya et McKelvey [Benbya& 06] notent également dans le contexte de la stratégie numérique qu'une synchronisation dynamique est nécessaire entre le business et l'IT pour assurer un avantage concurrentiel. D'autres travaux récents d'alignement ont également révélé que la stratégie est mise en œuvre au moyen d'une série de processus qui se déroulent au fil du temps. L'obtention et le maintien d'un bon alignement entre l'entreprise et les logiciels sous-jacents représentent un défi crucial puisque l'entreprise et les logiciels sont liés alors que les préoccupations et les intervenants métiers et en TI sont généralement différents.

L'évolution logicielle prend ainsi toute son importance dans la stratégie de toute entreprise. La problématique de l'alignement business-IT est particulièrement forte entre le niveau 2 et le niveau 3 d'un système d'information :

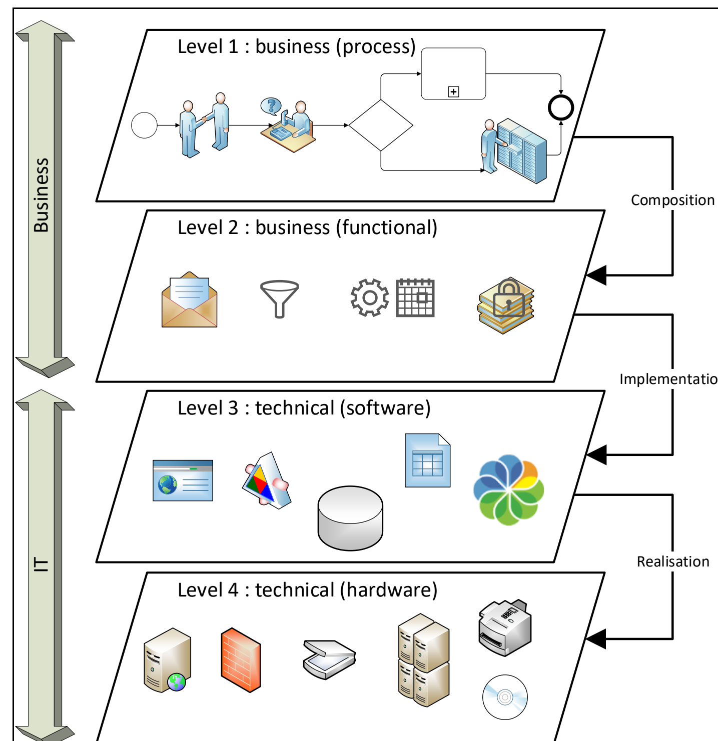


Figure 54 : les 4 niveaux d’un système d’information.

Yeow, Soh et Hansen dans leur article « *Aligning with new digital strategy: A dynamic capabilities approach* » [Yeow& 18] présentent une approche basée sur les capacités dynamiques d’une organisation à s’adapter. Elle se décline en trois phases : détection (*sensing*), capture (*seizing*) et transformer (*transforming*). Une ingénierie de l’évolution logicielle permettrait de se connecter aux bons endroits avec l’organisation pour apporter une assistance semi-automatisée de l’alignement.

Ainsi, ces trois phases de :

1. **Détection :** implique "l’identification, le développement, le co-développement et l’évaluation des opportunités technologiques en relation avec les besoins des clients".
2. **Capture :** implique la conception des nouvelles structures et nouveaux processus, la sélection parmi les différentes options et l’engagement.
3. **Transformation :** implique un renouvellement continu impliquant " l’alignement des actifs, le co-alignement, le réalignement et le redéploiement ".

Ce projet prend son ancrage dans ma collaboration avec Jean-Philippe Gouigoux, directeur technique de l’éditeur logiciel MGDIS, basé à Vannes. Nous travaillons actuellement sur le recensement d’anti-patterns d’alignement business-IT tirés de l’analyse de l’alignement d’une trentaine de systèmes d’information. Sur ce projet, des collaborations en interne au sein du LS2N sont également envisagées, notamment avec Hugo Brunelière (NaoMod) ainsi que Pascal André (AeLoS). Nous avons déjà eu l’occasion de travailler sur un contrat de collaboration avec Orange sur les problématiques de transformation numérique des entreprises.

d. Architecture logicielles continues et aide à la décision dans un environnement complexe et incertain

Ce projet en est au stade de la réflexion. Il se nourrira selon l'avancement des projet précédemment présentés. La problématique du *Continuous architecting* dans des contextes volatiles, incertains, complexes et hétérogènes impose un équilibre à trouver entre prévision (plutôt à court terme) et adaptabilité (plutôt à moyen, voire long termes), en sachant notamment exploiter la connaissance et savoir où l'exploiter. Ce projet pourrait être envisagé à plus long terme. Je le verrai toutefois se dérouler en parallèle et de manière transverse des 3 projets précédemment présentés. Ils pourraient servir de cas d'étude.

e. Collaborations scientifiques

J'aborde dans cette section les collaborations potentielles avec des équipes de recherche sur les aspects évolution et architectures logicielles :

- *Au niveau national :* Stéphane Ducasse et son équipe INRIA RMoD, notamment sur les aspects reengineering, software analysis, program visualization, reverse engineering, Lionel Seinturier et son équipe Spirals, du laboratoire CRIStAL à Lille, qui s'intéresse aux problèmes d'auto-adaptation pour les services distribués et les grands systèmes logiciels, l'équipe MAREL du LIRMM qui s'intéresse notamment à l'ingénierie de la réutilisation et de la variabilité ou encore l'équipe ArchWare de Flavio Oquendo à l'IRISA qui travaille entre autres sur la problématique d'évolution de systèmes-de-systèmes.
- *Au niveau international :* des échanges sont actuellement en cours pour monter des collaborations avec la Grande-Bretagne et le Québec (Canada) :
    - Grande-Bretagne : avec Joost Noppen, Principal Researcher à British Telecom, sur l'adoption et la migration de systèmes logiciels existants vers les micro-services. Nous nous appuyons sur les travaux de migration effectués avec Tom Mens.
    - Québec :
        - D'une part avec le laboratoire LATECE avec l'Université du Québec à Montréal (UQAM, Canada) impliquant H. Mili et N. Mouha ainsi que Y-G Guéhéneuc de Concordia et G. El Boussaidi de l'ETS (École de technologie supérieure). Il s'agit de mixer nos deux approches, fonctionnelle et technique, pour automatiser la détection des micro-services et identification de leur granularité :
        - D'autre part le laboratoire GEODES de l'Université de Montréal (Canada) avec H. Sahraoui et E. Syriani. Toujours dans le cadre de la migration automatisée, ce travail consistera à apprendre les processus de migration par l'exemple. Il s'agira d'utiliser des techniques de machine learning pour apprendre à partir de plusieurs migrations effectuées séparément et de manière had-hoc.

4. Perspectives vers la formation universitaire et le monde socio-économique

Un projet scientifique se doit de prendre en compte, outre la dimension purement scientifique, deux dimensions cruciales et auxquelles j'adhère : la formation et la collaboration avec le monde socio-économique :

4.1. Valorisation de la formation universitaire par la recherche

Lorsque l'on sait que 80% des métiers du futur dans les cinq années à venir n'existent pas, il me semble important de travailler au montage de nouvelles formations en concertation avec le monde

socio-économique. L'idée serait d'essayer d'anticiper sur des niches particulières les besoins à venir, mais cela reste insuffisant. L'enseignement supérieur doit former à accueillir et à savoir rebondir face à l'imprévisible et au non planifiable !

Mon principal dessein est d'impliquer et d'intégrer des enseignements liés aux travaux menés autour de la notion de service ainsi que ceux de mon projet à court terme :

1. Les formations à bac+1 jusqu'à bac+3 en montant des enseignements alliant théorie et cas pratiques émanant des entreprises.
2. Les masters recherche de manière à faire venir des enseignants-chercheurs de références ainsi que des intervenants d'entreprises, qui seraient couplées avec des séminaires de laboratoires ou lors de soutenances de thèses et de HDR.
3. Les masters recherche de manière que les étudiants puissent faire une partie de leur cursus à l'étranger, les préparant ainsi à la mobilité qui est de plus en plus exigées lors des recrutements.
4. La mise en place de formations co-habilitées entre des universités européennes et internationales.
5. Les doctorants en mettant en place un partenariat pour privilégier des postes de post-doctorants à l'étranger, et notamment dans des pays anglophones.
6. Enfin, travailler en étroite collaboration avec le monde socio-économique pour identifier et monter des formations adaptées à ses attentes.

Je ne perds pas de vue également l'importance de sensibiliser les étudiants et d'impacter les travaux aboutis, concrétisés et outillés au sein des formations initiales. Ainsi, l'introduction de techniques éprouvées et utilisées en industrie au sein des formations initiales permet de préparer les étudiants à certaines réalités et problèmes que posent les enjeux logiciels dans le milieu industriel. Il faut cependant anticiper, en concertation avec les acteurs et partenaires économiques, les besoins à venir.

4.2. Valorisation des collaborations avec le monde socio-économique

Cet aspect est important. En tant que chargée de mission du thème transverse EDFutur du LS2N et en tant que coordonatrice de la filière NUMERIC[22] de l'Université de Nantes, je suis appelée à fédérer les enseignants-chercheurs de la filière, à connaître et faire connaître les compétences du LS2N et des laboratoires et composantes de la filière, favoriser les liens et actions transverses et interdisciplinaires ainsi que représenter et promouvoir l'Université dans les réseaux socio-économiques (pôles de compétitivité, clusters, chambres consulaires…) et aider au pilotage stratégique sur l'implication de l'établissement dans les structures extérieures.

Cette valorisation avec le monde socio-économique doit se décliner en collaborations régionales, nationales, européennes et internationales mais également au travers de réseaux d'excellence. Elle passe par des collaborations scientifiques entre partenaires universitaires. Différents groupes de recherche s'intéressent à la problématique de l'évolution et de ce qui y a trait (maintenance, déploiement, traçabilité, passage à l'échelle, systèmes patrimoniaux …). Il est important que ces groupes ainsi que leurs membres actifs puissent travailler en synergie.

[22] https://www.univ-nantes.fr/innovez-grace-a-notre-recherche/filiere-numeric-numerique-usages-mathematiques-electronique-reseaux-informatique-communication-profitez-de-notre-expertise-pour-innover-1219647.kjsp

Chapitre 6.

Bibliographie

Références citées

[Alexander C. 96] C. Alexander, 'The Origins of Pattern Theoty, the Future of the Theory, and the Generation of a living World', Speech at OOPSLA 1996, San Jose, CA, 6-10 oct. 1996.

[Allen& 96] R. Allen et D. Garlan. 'The wright architectural specification language', Rapport technique CMU-CS-96-TBD, Carnegie Mellon University, School of Computer Science, 1996.

[Anquetil& 99] N. Anquetil et T. C. Lethbridge. Recovering software architecture from the names of source files. Journal of Software Maintenance, 11(3) :201–221, 1999.

[Anquetil& 07] N. Anquetil, K. de Oliveira, K.D. de Sousa, & M. G. B. Dias, Software maintenance seen as a knowledge management issue. Information and Software Technology, 49(5), 515-529. 2007.

[Ardagna& 07] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. Software Engineering, IEEE Transactions on, 33(6) :369–384, 2007.

[Arisholm & 04] E. Arisholm, L.C. Briand, A. Foyen, Dynamic coupling measurement for object-oriented software. IEEE Transactions on software engineering, 30(8), 491-506. 2004.

[Armbrust& 10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of Cloud computing. Communications of the ACM, 53(4) :50–58, 2010

[Arsanjani& 07] A. Arsanjani, L.J. Zhang, M. Ellis, A. Allam, and K. Channabasavaiah. S3 : A serviceoriented reference architecture. IT professional, 9(3) :10–17, 2007.

[Baghdadi 12] Y. Baghdadi, A methodology for Web services-based SOA realization', Int. Journal of Business Information Systems, Vol. 10, No. 3, pp.264–297, Inderscience Editors. 2012.

[Bain 08] S. L. Bain, 'Emergent Design: The Evolutionary Nature of Professional Software Development', mars 2008, Addison-Wesley Professional, ISBN: 0321509366.

[Bajaj& 06] S. Bajaj, D. Box, D. Chappell, F. Curbera, G. Daniels, P. Hallam-Baker, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, et al. Web services policy 1.2-framework (ws-policy). W3C Member Submission, 25 :12, 2006.

[Balalaie& 16] A. Balalaie, A. Heydarnoori, P. Jamshidi, Micro-services architecture enables devops: Migration to a cloud-native architecture. Ieee Software, 33(3), 42-52. 2016.

[Baldwin& 99] C.Y. Baldwin, K.B. Clark: Design Rules, vol. 1. MIT, Cambridge. 1999.

[Barais 05] O. Barais. Construire et Maîtriser l'Evolution d'une Architecture Logicielle à base de Composants, Thèse de Doctorat, Université des Sciences et Technologies de Lille, 2005.

[Barham& 03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, A., & A. Warfield, A. Xen and the art of virtualization. In ACM SIGOPS operating systems review (Vol. 37, No. 5, pp. 164-177). ACM. 2003.

[Barnes& 14] J.M. Barnes, D. Garlan, & B. Schmerl, Evolution styles: foundations and models for software architecture evolution. Software & Systems Modeling, 13(2), 649-678. 2014.

[Bass& 98] Bass L., P. Clements et R. Kazman. Software Architecture In Practice. Addison-Wesley Publishing, 1998.

[Bayer& 06] J. Bayer, S. Gerard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.P. Thibault, and T. Widen. Consolidated product line variability modeling. 2006

[Belhajjame& 05] K. Belhajjame, G. Vargas-Solar, C. Collet, Pyros - An environment for building and orchestrating open services. Proceedings of the IEEE International Conference on Services Computing, ICSC'03, pp.155–164, USA. 2005.

[Bellwood& 02] T. Bellwood, L. Clément, D. Ehnebuske, A. Hately, M. Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, et al. Uddi version 3.0. Published specification, Oasis, 5 :16–18, 2002.

[Benbya& 06] Benbya, H., McKelvey, B.,. Using coevolutionary and complexity theories to improve IS alignment: a multi-level approach. J. Inform. Technol. 21 (4), 284–298. 2006.

[Bencomo& 08] N. Bencomo, P. Sawyer, G. Blair, and P. Grace. Dynamically adaptive systems are product lines too : Using model-driven techniques to capture dynamic variability of adaptive systems. In 2nd International Workshop on Dynamic Software Product Lines (DSPL 2008), Limerick, Ireland, volume 38, page 40, 2008.

[Bennett& 00] K.H. Bennett, & V.T. Rajlich, Software maintenance and evolution: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 73-87). ACM. 2000.

[Bezemer& 10] C.P. Bezemer and A. Zaidman. Multi-tenant SaaS applications : maintenance dream or nightmare ? In Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), pages 88–92. ACM, 2010.

[Bieberstein& 08] N. Bieberstein, R. Laird, K. Jones, T. Mitra (May 5, 2008). Executing SOA: A Practical Guide for the Service-Oriented Architect. IBM Press.

[Bieman& 95] J.M. Bieman, B.K. Kang, Cohesion and reuse in an object-oriented system. ACM SIGSOFT Software Engineering Notes, 20(si), 259-262. 19

[Binns& 96] P. Binns, M. Englehart, M. Jackson, S. Vestal: Domain-specific software architectures for guidance, navigation, and control. International Journal of Software Engineering and Knowledge Engineering 6, 2,1996.

[Boag& 03] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0 : An xml query language. W3C working draft, 12, 2003.

[Bosch 01] J. Bosch. Software product lines : organizational alternatives. In Proceedings of the 23rd International Conference on Software Engineering, pages 91–100. IEEE Computer Society, 2001

[Brand& 83] D. Brand & P. Zafiropulo. On communicating finite-state machines. Journal of the ACM (JACM), 30(2), 323-342. 1983.

[Brooks 95] F.P. Brooks, The Mythical Man-Month: Essays on Software Engineering. ISBN 0-201-83595-9. 1975, 2nd ed. 1995.

[Bruneton& 06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, & J.B. Stefani, The fractal component model and its support in java. Software: Practice and Experience, 36(11-12), 1257-1284., 2006.

[Buyya& 09] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms : Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems, 25(6) :599–616, 2009

[Canal& 06] C. Canal, J. Manuel, & M.P. Poizat, Software adaptation. In in L'objet, 12 (1): 9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities., 2006.

[Canfora& 11] G. Canfora, M. Di Penta, & L. Cerulo, Achievements and challenges in software reverse engineering. Commun. ACM, 54(4), 142-1, 2011.

[Casati& 01] F. Casati F., M.C. Shan M.C., Dynamic and adaptive composition of e-services, Journal of Information Systems, Vol. 26, n°3, pp. 143-163. Elsevier. 2001.

[Cassidy 16] A. Cassidy, A practical guide to information systems strategic planning. CRC press.2016.

[Cetina& 08] C. Cetina, J. Fons, and V. Pelechano. Applying software product lines to build autonomic pervasive systems. In Software Product Line Conference, 2008. SPLC'08. 12th International, pages 117–126. IEEE, 2008.

[Chaffin 89] R. Chaffin, The nature of semantic relations: a comparison of two approaches. In Relational models of the lexicon (pp. 289-334). Cambridge University Press. 1989.

[Chaki& 04] S. Chaki, N. Sharygina, N. Sinha,Verification of evolving software. In: Proceedings of the Workshop on Specification and Verification of Component Based Systems (SAVCBS 2004), pp. 55–61. Iowa State University, Ames. 2004.

[Chan& 97] Y.E. Chan, S.L. Huff, D. W. Barclay, D.G. Copeland, Business strategic orientation, IS strategic orientation, and strategic alignment. Information systems research, 8(2), 125-150. 1997.

[Chang& 07] S.H. Chang and S.D. Kim. A variability modeling method for adaptable services in service-oriented computing. In Software Product Line Conference, 2007. SPLC 2007. 11th International, pages 261–268. Ieee, 2007

[Chapin& 01] N. Chapin, J.E. Hale, K.M. Khan, J.F. Ramil, & W.G. Tan, Types of software evolution and software maintenance. Journal of software maintenance and evolution: Research and Practice, 13(1), 3-30. 2001.

[Chapin 04] N. Chapin, 'Agile methods - contributions in software evolution', Software Maintenance, 2004. Proceedings 20th IEEE International Conference, 11-14 Sept. Page(s): 522 – ICSM 2004. ISBN 0-7695-2213-0. 2004

[Charfi& 07] A. Charfi and M. Mezini. Ao4bpel : An aspect-oriented extension to BPEL. World Wide Web, 10(3) :309–344, 2007.

[Chebbi 07] I. Chebbi I., CoopFlow : Une approche pour la coopération ascendante dans les entreprises virtuelles. Thèse de doctorat. Institut National des Telecoms, France. 2007.

[Chikofsky& 90] E.J. Chikofsky, J.H. Cross "Reverse Engineering and Design Recovery: A Taxonomy in IEEE Software". IEEE Computer Society: 13–17. 1990.

[Chinnici& 07] R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web services description language (wsdl) version 2.0 part 1 : Core language. W3C Recommendation, 26, 2007.

[Chong& 06] F. Chong and G. Carraro. Architecture strategies for catching the long tail. MSDN Library, Microsoft Corporation, pages 9–10, 2006.

[CIGREF 03] https://www.cigref.fr/cigref_publications/RapportsContainer/Parus2003/2003_-_Accroitre_l_agilite_du_systeme_d_information_web.pdf

[Curbera& 02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web : an introduction to soap, wsdl, and uddi. Internet Computing, IEEE, 6(2) :86–93, 2002.

[Curbera& 05] F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. Web services platform architecture : SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more. Prentice Hall PTR, 2005.

[Dashofy& 01] E.M. Dashofy, A. van der Hoek et R.N. Taylor, 'A highly-extensible, xml-based architecture description language.' in Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001), Amsterdam, Netherlands, 2001.

[Dayal& 88] U. Dayal, A. Buchmman et D. Mccarthy. Rules are objects too: a knowledge model for an active object-oriented database systems. Lecture Notes in Computer Science 334, pages 129–143, 1988.

[Dinkelaker& 10] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In 5th Domain-Specific Aspect LanguagesWorkshop, 2010.

[Diot& 99] Diot, C., & Gautier, L. (1999). A distributed architecture for multiplayer interactive applications on the Internet. IEEE network, 13(4), 6-15.

[Du& 10] J. Du, H. Wen, Z. Yang. Research on data layer structure of multi-tenant e-commerce system. In Industrial Engineering and Engineering Management (IE&EM), 2010 IEEE 17Th International Conference on, pages 362–365. IEEE, 2010.

[Dua& 14] R. Dua, A. R. Raja,, D. Kakadia, Virtualization vs containerization to support paas. In 2014 IEEE International Conference on Cloud Engineering (pp. 610-614). IEEE. 2014.

[Edwards& 01] P. Edwards, M. Peters, and G. Sharman. The effectiveness of information systems in supporting the extended supply chain, Journal of business logistics, Wiley Online Library, vol. 22, Number 1, pages 1--27, 2001.

[Esper 10] A. Esper, Integration des approches SOA et orientée objet pour modéliser une orchestration cohérente de services. Thèse de doctorat, INSA, Lyon, France. 2010.

[Evans 03] E. Evans, Domain Driven Design, Tackling Complexity in the Heart of Software. 2003.

[Fensel& 03] D. Fensel, E. Motta, F. van Harmelen, V.R. Benjamins, M. Crubezy, S. Decker, M. Gaspari, R. Groenboom, W. Grosso, M. Musen, E. Plaza, G. Schreiber, R. Studer, B. Wielinga : The unified problem-solving method development language upml. Knowl. Inf. Syst. 5, 1, 83–131. 2003.

[Fenton& 14] N. Fenton, J. Bieman, Software metrics: a rigorous and practical approach. CRC press. 2014.

[Fernández& 19] DM Fernández, W Böhm, A Vogelsang, J Mund , M. Broy, M., Kuhrmann, T. Weyer. Artefacts in software engineering: a fundamental positioning. Software & Systems Modeling, 1-10. 2019.

[Feuerlicht 10] G. Feuerlicht. Next generation soa : Can soa survive Cloud computing ? pages 19–29, 2010

[Fielding 00] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[Fleurey& 07] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, & J.M. Jézéquel, Model-driven engineering for software migration in a large industrial context. In International Conference on Model Driven Engineering Languages and Systems (pp. 482-497). Springer, Berlin, Heidelberg. 2007.

[Fielding 00] Fielding, R. (2000). Representational state transfer. Architectural Styles and the Design of Netowork-based Software Architecture, 76-85.

[Finkelstein& 94] A. C. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency handling in multiperspective specifications," IEEE Transactions on Software Engineering, vol. 20, no. 8, pp. 569–578, 1994.

[Fowler 99] M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Weslrey, 1999. ISBN 0-201-48567-2.

[Fowler& 14] M. Fowler, J. Lewis, J. Micro-services. ThoughtWorks. http://martinfowler. com/articles/micro-services. Html. 2014.

[Francesco& 18] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards micro-service architectures: an industrial survey. In 2018 IEEE International Conference on Software Architecture (ICSA) (pp. 29-2909). IEEE. 2018.

[Gallier& 14] R.D. Galliers, D.E. Leidner, Strategic information management: challenges and strategies in managing information systems. Routledge. 2014.

[Gamma& 94] E. Gamma, R. Helm, R. Johnson and J. Vlissides, 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley, ISBN: 0201633612. 1994.

[Garlan& 94] D. Garlan, R. Allen, J. Ockerbloom, Exploiting style in architectural design environments. SIGSOFT Softw. Eng. Notes 19, 5, 175–188. 1994.

[Garlan& 95] D. Garlan, R. Allen, J. Ockerbloom, Architectural mismatch or why it's hard to build systems out of existing parts. In ICSE '95: Proceedings of the 17thinternational conference on Software engineering (New York, NY, USA), ACM, pp. 179–185. 1995.

[Garlan, 95] D. Garlan. What is style? In Proceedings of the Dagstuhl Workshop on Software Architecture, Saarbruecken, Germany, February 1995.

[Garlan& 97] D. Garlan, R. Monroe et D.Wile. Acme : An architecture description interchange language., Dans Proceedings of CASCON'97, Toronto, Canada, 1997.

[Garlan, 00] D. Garlan, Software architecture and object-oriented systems. In Proceedings of the IPSJ Object-Oriented Symposium 2000, August 2000.

[Garlan& 00] D. Garlan, R. Monroe et D.Wile, 'Acme : An architecture description of component-bases systems.', Fondations of Component-Based Systems, Leavens Gray et Sitaraman Murali (Réd.), pages 47–68, 2000.

[Garlan 08] D. Garlan, "Evolution Styles Formal Foundations and Tool Support for Software Architecture Evolution," Tech. report. CMU-CS-08-142, CMU, 2008.

[Garlan& 09] D. Garlan, J. M. Barnes, B. Schmerl, & O. Celiku, Evolution styles: Foundations and tool support for software architecture evolution. In Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (pp. 131-140). IEEE. 2009

[Geelan 09] J. Geelan. Twenty one experts define Cloud computing. Cloud Computing Journal, 4 :1–5, 2009.

[Ghezzi& 91] C. Ghezzi, M. Jazayeri, D. Mandrioli, Fundamentals of Software Engineering. Prentice Hall, Upper Saddle River, 1991.

[Gilb 81] T. Gilb, "Evolutionary development", ACM Sigsoft, Software Engineering Notes, Vol 6, n° 2, avril 1981.

[Grunske 05] L. Grunske, Formalizing Architectural Refactorings as Graph Transformation Systems, in: Int'l Conf. SNPD and ACIS Int'l Workshop SAWN, 324–329, 2005.

[Guo& 07] C.J. Guo, W. Sun, Y. Huang, Z.H. Wang, and B. Gao. A framework for native multitenancy application development and management. In E-Commerce Technology and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and EServices, 2007. CEC/EEE 2007. The 9th IEEE International Conference on, pages 551–558. IEEE, 2007.

[Gustavo& 04] A. Gustavo, F. Casati, H. Kuno, H., V. Machiraju, Web services: concepts, architectures and applications. Springer Verlag, Germany. 2004.

[Hacigumus& 02] H. Hacigumus, B. Iyer, S. Mehrotra. Providing database as a service. In Data Engineering, 2002. Proceedings. 18th International Conference on, pages 29–38. IEEE, 2002.

[Hancox& 00] M. Hancox, R. Hackney. It outsourcing : frameworks for conceptualizing practice and perception. Information Systems Journal, 10(3) :217–237, 2000.

[Harman 07] M. Harman, The current state and future of search based software engineering. In Future of Software Engineering, pp. 342–357. IEEE. 2007.

[Harris& 95] D.R. Harris, H.B. Reubenstein, A.S. Yeh, " Reverse Engineering to the Architectural Level ", Proc. of ICSE, ACM, Inc., p. 186-195, 1995.

[Heinemann 01] G. Heinemann, et W. Councill. Component-based software engineering. Addison Wesley. 2001.

[Heorhi 12] R. Heorhi, Service Composition in Dynamic Environments: From Theory to Practice, Phd Thesis, University of Trento. 2012.

[Hirschheim& 01] R. Hirschheim, R. Sabherwal, Detours in the path toward strategic information systems alignment. California management review, 44(1), 87-108. 2001.

[Hoare 95] C. Hoare. Communicating sequential processes., 1995.

[Hock-Koon 11] A. Hock-Koon. Contribution à la compréhension et à la modélisation de la composition et du couplage faible de services dans les architectures orientées services. PhD thesis, UFR Sciences et Techniques, Université de Nantes, 2011.

[Hoeck 05] A.V.D Hoek, M. Rakic, R.Roshandel et N. Medvidovic.Taming architectural evolution. Dans Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9, pages 1–10, Vienna, Austria, 2005.

[Horlach& 16] B. Horlach, P. Drews, & I. Schirmer. Bimodal IT: Business-IT alignment in the age of digital transformation. Multikonferenz Wirtschaftsinformatik (MKWI), 1417-1428. 2016.

[IEEE 90] IEEE Standard Group Glossary of Software Engineering Terminolgy (IEEE Std. 610.12-1990). IEEE, New York, 1990.

[Jacobson& 97] I. Jacobson, M. Griss, et P. Jonsson. Software Reuse. Addison Wesley/ACM Press, 1997.

[Jansen& 10] S. Jansen, G.J. Houben, and S. Brinkkemper. Customization realization in multi-tenant web applications : case studies from the library sector. Web Engineering, pages 445–459, 2010.

[Jaramillo& 16] D. Jaramillo, D.V. Nguyen, R. Smart, Leveraging micro-services architecture by using Docker technology. In SoutheastCon 2016 (pp. 1-5). IEEE. 2016

[Jazayeri& 00] M. Jazayeri, A. Ran et F. van der Linden. Software architecture for product families: principles and practice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000

[Jazayeri 05] M. Jazayeri. Species evolve, individuals age. In Principles of Software Evolution, Eighth International Workshop on, pages 3-9, 2005.

[Kabbedijk& 11] J. Kabbedijk and S. Jansen. Variability in multi-tenant environments: architectural design patterns from industry. Advances in Conceptual Modeling. Recent Developments and New Directions, pages 151–160, 2011.

[Kang& 10] S. Kang, J. Myung, J. Yeon, S.W. Ha, T. Cho, J. Chung, S.G. Lee. A general maturity model and reference architecture for SaaS service. In Database Systems for Advanced Applications, pages 337–346. Springer, 2010.

[Karpovsky& 15] Karpovsky, A., Galliers, R.D.. Aligning in practice: from current cases to a new agenda. J. Inform. Technol. 30 (2), 136–160. 2015.

[Kazman& 98] R. Kazman, S. Woods et S. Carrière. Requirements for integrating software architecture and reengineering models : Corum ii.In Working Conference on Reverse Engineering, pages 154–163. IEEE Computer Society Press, 1998.

[Kazman& 99] R. Kazman et S. J. Carrière. Playing detective : Reconstructing software architecture from available evidence. Automated Software Engg., 6(2) :107–138, 1999.

[Ketler& 93] K. Ketler, J.Walstrom. The outsourcing decision. International journal of information management, 13(6) :449–459, 1993.

[Khajeh-Hosseini 10] A. Khajeh-Hosseini, I. Sommerville, I. Sriram. Research challenges for enterprise Cloud computing. arXiv preprint arXiv :1001.3257, 2010.

[Kim& 89] W. Kim, E. Ber3ztino et J.F. Garza, Composite objects revisited. International Conference on Management of Data, pages 337–347, 1989.

[Koning& 09] M. Koning, C. Sun, M. Sinnema, and P. Avgeriou. Vxbpel : Supporting variability for web services in bpel. Information and Software Technology, 51(2) :258–269, 2009.

[Koschke 00] R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. Thèse de Doctorat, University of Stuttgart, 2000

[Koziolek 10] H. Koziolek, Towards an architectural style for multi-tenant software applications. In Gregor Engels, Markus Luckey, and Wilhelm Schäfer, editors, Software Engineering, volume 159 of LNI, pages 81–92. GI, 2010.

[Koziolek 11] H. Koziolek. The sposad architectural style for multi-tenant software applications. In Software Architecture (WICSA), 2011 9th Working IEEE/IFIP Conference on, pages 320–327. IEEE, 2011.

[Kramer 07] J. Kramer 'Is abstracting the key of computing?', Communications of the ACM, april. Vol. 50, n° 4. Pages 36 – 42. 2007

[Krebs& 12] R. Krebs, C. Momm, S. Konev. Architectural concerns in multitenant SaaS applications. In Proceedings of the 2nd International Conference on Cloud Computing and Service Science (CLOSERŠ12), SciTePress, 2012

[Krikhaar 97] L. Krikhaar. Reverse architecting approach for complex systems. In ICSM '97: Proceedings of the International Conference on Software Maintenance, pages 4–11, Washington, DC, USA,. IEEE Computer Society, 1997.

[Kruchten& 06] P. Kruchten, H. Obbink, J. Stafford: The past, present, and future for software architecture. IEEE Softw. 23, 2. 2006.

[Kwok& 08] T. Kwok, T. Nguyen, and L. Lam. A software as a service with multi-tenancy support for an electronic contract management application. In Services Computing, 2008. SCC'08. IEEE International Conference on, volume 2, pages 179–186. IEEE, 2008

[Le Goaer 09] O. Le Goaer, Styles d'évolution dans les architectures logicielles (Doctoral dissertation), 2009.

[Le Metayer 98] D. Le Metayer, Describing software architecture styles using graph grammars, IEEE Trans. Softw. Eng. 24 (7), 521–533. 1998

[Laarhoven& 87] P. J. M. Laarhoven et E. H. L. Aarts, réds. Simulated annealing : theory and applications. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[Lagaisse& 06] B. Lagaisse, W. Joosen. True and transparent distributed composition of aspectcomponents. In Middleware 2006, pages 42–61. Springer, 2006.

[Laplante& 08] P.A. Laplante, J. Zhang, and J. Voas. What's in a name ? distinguishing between saas and soa. IT Professional, 10(3) :46–50, 2008.

[Lee& 95] Y. Lee, B. Liang, S. Wu, F. Wang, " Measuring the Coupling And Cohesion of an Object-Oriented Program Based on Information Flow ", ICSQ'95, p. 81-90, 1995.

[Li Heng& 12] Y. D. Li Heng, Z. Xiaohong. A new meta-data driven data-sharing storage model for saas. IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 6, No 1, November 2012 ISSN (Online) : 1694-0814, 2012.

[Li& 93] W. Li, S. Henry, " Object Oriented Metrics that Predict Maintainability ", Journal of Systems and Software, vol. 223, p. 111-122, 1993.

[Liu& 10] W Liu, B Zhang, Y Liu, D Wang, Y. Zhang. New model of SaaS : Saas with tenancy agency. In Advanced Computer Control (ICACC), 2010 2nd International Conference on, volume 2, pages 463–466. IEEE, 2010.

[Lehman 78] M.M. Lehman, Laws of program evolution--rules and tools for programming management, in: Proc. of the Infotech State of the Art Conference, Why Software Projects Fail, 1978;

[Lehman& 85] M. M. Lehman, L. A. Belady, Program evolution: processes of software change, Academic Press Professional, Inc., San Diego, CA, 1985,

[Lehman 96] M. M. Lehman, Laws of software evolution revisited. In European Workshop on Software Process Technology (pp. 108-124). Springer, Berlin, Heidelberg. October, 1996.

[Lehman 05] M.M. Lehman, "Approach to a Theory of Software Evolution," IWPSE, p. 135, Eighth International Workshop on Principles of Software Evolution (IWPSE'05), 2005

[Li& 93] W. Li, S. Henry, " Object Oriented Metrics that Predict Maintainability ", Journal of Systems and Software, vol. 223, p. 111-122, 1993.

[Luer& 02] C. Luer, et A. van der Hoek. Composition environments for deployable software components. Technical report. 2002.

[Luckham& 95] D.C Luckham, J.J Kenney, L.M. Augustin, J. Vera, D. Bryan, & W. Mann. Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering, 21(4), 336-354. 1995.

[Luftman 04] J. Luftman, Assessing business-IT alignment maturity. Strategies for information technology governance, 4, 99. 2004.

[Luftman& 17] J. Luftman, K. Lyytinen, & T.B. Zvi, Enhancing the measurement of information technology (IT) business alignment and its influence on company performance. Journal of Information Technology, 32(1), 26-46. 2017.

[Magee& 95] J. Magee, N. Dulay, S. Eisenbach et J. Kramer. Specification and analysis of system architectures using rapide. Dans Proceedings of the fifth European Software Engineering conference, pages 336–355, Barcelona, Spain, 1995.

[Madhavji& 06] N. H. Madhavji, J. Fernandez-Ramil and D. Perry, "Software Evolution and Feedback: Theory and Practice, ISBN 0470871806, John Wiley & Sons Ed. 2006.

[Marz& 15] N. Marz, J. Warren, Big Data: Principles and best practices of scalable realtime data systems. Manning Publications Co. 2015.

[McBride 07] M.R. McBride, 'The Software Architect', Communications of the ACM, Vol. 50, n° 5. Pages: 75-81. 2007.

[McVeigh& 06] A. McVeigh, J. Kramer, J. Magee, Using resemblance to support compo- nent reuse and evolution, in: Proc. Conf. Specification and Verification of Component-based Systems, ACM, ISBN 1-59593-586-X, 49–56, 2006.

[Medvidovic & 96] N. Medvidovic, R. Taylor et E. Whitehead. Formal modeling of software architectures at multiple levels of abstraction. Dans Proceedings of the 1996 California Software Symposium, pages 28–40, Los Angeles, CA, 1996.

[Medvidovic & 98] N. Medvidovic, D.S. Rosenblum et R.N. Taylor. Type theory for software architectures. Dans Technical Report, UCI-ICS-98-14, University of California, Irvine, 1998.

[Medvidovic & 99] N. Medvidovic, D.S. Rosenblum et R.N. Taylor. A language and environment for architecture-based software development and evolution. Dans Proceeding of the 21st international conference on Software engineering, (ICSE'99), pages 44–53, 1999.

[Medvidovic & 00] N. Medvidovic, R.N. Taylor: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26, 1, 70–93. 2000.

[Medvidovic & 06] N. Medvidovic, V. Jakobac, " Using software evolution to focus architectural recovery ", Automated Software Engineering, vol. 13, p. 225-256, 2006.

[Meijler& 98] T. D. Meijler and O. Nierstrasz. Beyond Objects: Components. In M. Papazoglou and G. Schlageter, editors, Cooperative Information Systems: Trends and Directions. Academinc Press, 1998.

[Mell& 11] P. Mell and T. Grance. The nist definition of Cloud computing. National Institute of Standards and Technology special publication, 800 :145, 2011.

[Mens 02] T. Mens "A State-of-the-Art Survey on Software Merging", IEEE Transactions on Software Engineering archive, Volume 28 , Issue 5, pages: 449 – 462, , ISSN:0098-5589. 2002.

[Mens& 01] T. Mens and M. Wermelingern, "Formal foundations of software evolution: workshop report", SIGSOFT Softw. Eng. Notes, vol. 26, N° 4, ISSN 0163-5948, pages 27--29, ACM, New York, NY, USA. 2001.

[Mens& 03] T. Mens, J. Buckley, M. Zenger, & A. Rashid, A. Towards a taxonomy of software evolution. In Proceedings of the International Workshop on Unanticipated Software Evolution, 2003.

[Mens& 04] T. Mens, & T. Tourwé, A survey of software refactoring. *IEEE Transactions on software engineering*, *30*(2), 126-139, 2004.

[Mens& 05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, M. Jazayeri, Challenges in software evolution, in: IWPSE, 2005.

[Mens& 08] T. Mens, S. Demeyer Eds. 'Software Evolution', XVIII, 347 p. 100 illus., Hardcover, ISBN: 978-3-540-76439-7, 2008.

[Metzger& 07] A. Metzger, P. Heymans, K. Pohl, P.Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines : A separation of concerns, formalization and automated analysis. In Requirements Engineering Conference, 2007. RE'07. 15th IEEE International, pages 243–253. IEEE, 2007.

[Meyer 88] B. Meyer "Object-Oriented Software Construction" International Series in Computer Science. Prenctice Hall, 1988.

[Mietzner& 08a] R. Mietzner, F. Leymann, and M.P. Papazoglou. Defining composite configurable SaaS application packages using sca, variability descriptors and multi-tenancy patterns. In Internet andWeb Applications and Services, 2008. ICIW'08. Third International Conference on, pages 156–161. IEEE, 2008.

[Mietzner& 08] R. Mietzner and F. Leymann. Generation of BPEL customization processes for saas applications from variability descriptors. In Services Computing, 2008. SCC'08. IEEE International Conference on, volume 2, pages 359–366. IEEE, 2008

[Mietzner& 09] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, pages 18–25. IEEE Computer Society, 2009.

[Mietzner 10] R. Mietzner. A method and implementation to define and provision variable composite applications, and its usage in Cloud computing. PhD thesis, Stuttgart University, 2010.

[Motta 00] E. Motta, The Knowledge Modeling Paradigm in Knowledge Engineering, vol. 1. World Scientific Publishing Co., ch. Handbook of Software Engineering and Knowledge Engineering. 2000.

[Muller& 00] Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M. A., Tilley, S. R., & Wong, K. Reverse engineering: a roadmap. In Proceedings of the Conference on the Future of Software Engineering (pp. 47-60). ACM, 2000.

[Namiot& 14] D. Namiot, M. Sneps-Sneppe, On micro-services architecture. International Journal of Open Information Technologies. 2014.

[Namjoshi& 09] J. Namjoshi and A. Gupte. Service oriented architecture for cloud based travel reservation software as a service. In Cloud Computing, 2009. CLOUD'09. IEEE International Conference on, pages 147–150. IEEE, 2009.

[Newcomer 05] Newcomer, Eric, Lomow, Greg "Understanding SOA with Web Services". Addison Wesley. ISBN 0-321-18086-0, 2005.

[Newman 15] Newman, S. (2015). Building micro-services: designing fine-grained systems. " O'Reilly Media, Inc.".

[Noppen& 10] J. Noppen et D. Tamzalit. ETAK : Tailoring Architectural Evolution by (re-)using Architectural Knowledge. In Fifth Workshop on SHAring and Reusing architectural Knowledge, pages 60–68, Cap Town, Afrique Du Sud, (SHARK 2010), ICSE 2010.

[OMG 03] Object Management GROUP. Uml 2.0 infrastructure specification, technical report ptc/03-09-15. 2003.

[OMG 07] Object Management Group, "Unified Modeling Language: Infrastructure version 2.1.2," 2007.

[Pahl& 06] C. Pahl and Y. Zhu. A semantical framework for the orchestration and choreography of web services. Electronic Notes in Theoretical Computer Science, 151(2) :3–18, 2006.

[Papazoglou 03] M.P. Papazoglou. Service-oriented computing : Concepts, characteristics and directions. In Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on, pages 3–12. IEEE, 2003.

[Papazoglou& 07] M.P. Papazoglou and W.J. Van Den Heuvel. Service oriented architectures : approaches technologies and research issues. The VLDB journal, 16(3) :389–415, 2007

[Parnas 72] D.L. Parnas, Information distribution aspects of design methodology. In: Proceedings of IFIP Congress '71, pp. 339–344. North-Holland, Amsterdam,1972.

[Parnas 94] D.L. Parnas, 'Software aging, in proceedings IEEE ICSE, pages 279-287., IEEE Computer Society Press, 1994.

[Pedraza 09] F.G. Pedraza, un canevas extensible pour la construction d'applications orientées procédé. Thèse de doctorat, Universite Joseph Fourier - GRENOBLE I. , 2009

[Peltz 03] C. Peltz. Web services orchestration and choreography. Computer, 36(10) :46–52, 2003.

[Perry& 92] D.E. Perry et A.L. Wolf. Foundations for the study of software architecture. Software Engineering Notes, ACM SIGSOFT, 17. 1992.

[Pervez& 10] Z. Pervez, S. Lee, Y.K. Lee. Multi-tenant, secure, load disseminated saas architecture. In Advanced Communication Technology (ICACT), 2010 The 12th International Conference on, volume 1, pages 214–219. IEEE, 2010.

[Pigoski 96] Pigoski, T. M. Practical software maintenance: best practices for managing your software investment. Wiley Publishing, 1996.

[Pilioura& 01] T. Pilioura, T., A. Tsalgatidou, E-services: Current technology and open issues. In International Workshop on Technologies for E-Services (pp. 1-15). Springer, Berlin, Heidelberg. 2001.

[Pohl& 05] K. Pohl, G. Bockle, and F. Van Der Linden. Software product line engineering: foundations, principles, and techniques. Springer-Verlag New York Inc, 2005.

[Polet& 07] D. Pollet, S. Ducasse, L. Poyet, I. Alloui, S. Cimpan et H. Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In Proc. of the CSMR, pages 137–148. IEEE, 2007.

[Rimal& 09] BP. Rimal, E. Choi, I. Lumb, A taxonomy and survey of cloud computing systems. In INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on, pages 44–51. Ieee, 2009.

[Riva 00] C. Riva. Reverse architecting : An industrial experience report, In WCRE '00 : Proceedings of the Seventh Working Conference on Reverse Engineering. (WCRE'00), page 42, Washington, DC, USA,. IEEE Computer Society. 2000.

[Ross& 04] J.W. Ross and G. Westerman. Preparing for utility computing : The role of it architecture and relationship management. IBM systems journal, 43(1) :5–19, 2004.

[Sauser 09] B. Sauser, « Reprogramming Satellites during Flight », MIT Technology Review, July 2009.

[Schroeter& 12] J. Schroeter, S. Cech, S. Götz, C. Wilke, and U. Aßmann. Towards modeling a variable architecture for multi-tenant saas-applications. In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, pages 111–120. ACM, 2012.

[Sengupta& 11] B. Sengupta and A. Roychoudhury. Engineering multi-tenant software-as-a-service systems. In Proceeding of the 3rd international workshop on Principles of engineering service-oriented systems, pages 15–21. ACM, 2011.

[Shah& 10] D. Shah, M. Agarwal, M. Mehra and A. Mangal, "Global SOA: RSS-Based Web Services Repository and Ranking," 2010 Fifth International Conference on Internet and Web Applications and Services, Barcelona, 2010, pp. 256-261.

[Shaw& 95] M. Shaw, R. DeLine, R., D.V. Klein, T.L. Ross, D.M. Young & G. Zelesnik. Abstractions for software architecture and tools to support them. IEEE transactions on software engineering, 21(4), 314-335, 1995.

[Shaw& 96] M. Shaw, D. Garlan, 'Software Architecture: Perspectives on an Emerging Discipline'. Prentice Hall, Paperback, Published, 240 pages, ISBN 0131829572. 1996

[Shaw& 06] M. Shaw & P. Clements, The golden age of software architecture. IEEE Software, 2(23), 31–39. doi:10.1109/MS.2006.58, 2006.

[Sheng& 02] Q.Z. Sheng, B. Benatallah, B., M. Dumas, E. O.I-Yan Mak , SELF-SERV: a platform for rapid composition of web services in a peer-to-peer environment. Proceedings of VLDB '02: the 28th international conference on Very Large Data Bases. 2002.

[Smeda& 05] A. Smeda, M. Oussalah, T. Khammaci, MADL: Meta Architecture Description Language, in: SERA, 152–159, 2005.

[Smeda& 08] A. Smeda, M.C. Oussalah, T. Khammaci, My architecture: a knowledge representation meta-model for software architecture. International Journal of Software Engineering and Knowledge Engineering 18, 7, 877–894, 2008.

[Smeda 06] A. Smeda, Contribution à l'élaboration d'une métamodélisation de description d'architecture logicielle (Doctoral dissertation, Nantes). 2006.

[Sommerville 07] I. Sommerville, "Software engineering", International Computer Science Series, 2007.

[Subashini& 11] S Subashini and V Kavitha. A survey on security issues in service delivery models of Cloud computing. Journal of Network and Computer Applications, 34(1) :1–11, 2011.

[Sultan 11] N. A. Sultan. Reaching for the "Cloud": How SMEs can manage. International Journal of Information Management, 31(3) :272–278, 2011.

[Sun& 10] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello. Modeling and managing the variability of web service-based systems. Journal of Systems and Software, 83(3) :502– 516, 2010.

[Syswerda 89] G. Syswerda "Uniform Crossover in Genetic Algorithms" dans les actes de Intl. Conf. On Genetic Algorithms, pages 2-9, 1989.

[Szyperski 97] C. Szyperski, 'Component software: Beyond object-oriented programming.' Addison Wesley Professional, , ISBN-10: 0-201-17888-5. 1997

[Szyperski 98] C. Szyperski, Component Software. ISBN : 0-201-17888-5. Addison-Wesley, 1998.

[Teway& 13] Tewary A., Kosalge P., Implementing service oriented architecture – a case study. Int. Journal of Business Information Systems, Vol. 14, no. 2, pp.164 –181, Inderscience. 2013

[Tograph& 08] B. Tograph and Y.R. Morgens. Cloud computing. Communications of the ACM, 51(7), 2008.

[Trinidad& 07] P. Trinidad, A. Ruiz-Cortés, J. Pena, and D. Benavides. Mapping feature models onto component models to build dynamic software product lines. In International Workshop on Dynamic Software Product Line, DSPL, 2007

[Tröls& 19] M. A. Tröls, A. Mashkoor, and A. Egyed, "Live and global consistency checking in a collaborative engineering environment," in The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), pp. 1762 – 1771, ACM, 2019.

[Truyen& 01] E. Truyen, B. Vanhaute, B. Joosen, P. Verbaeton. Dynamic and selective combination of extensions in component-based applications. In Proceedings of the 23rd International Conference on Software Engineering, pages 233– 242. IEEE Computer Society, 2001.

[Tzerpos& 96] V. Tzerpos, R. Holt, " A hybrid process for recovering software architecture ", Proc. of the conf. of the Centre for Advanced Studies on Collaborative Research, CASCON, p. 1-6, 1996.

[Vaquero& 08] L.M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds : towards a Cloud definition. ACM SIGCOMM Computer Communication Review, 39(1) :50– 55, 2008.

[Varma& 07] P. Varma, A. Anand, D. Pazel, & B. Tibbitts. *U.S. Patent Application No. 11/175,741*. 2007.

[Van der Aalst 99] W. M. P, Van Der Aalst, Process oriented architectures for electronic commerce and interogranizational workflow. Journal of Information systems, Vol. 24, n°9. 1999.

[Van der Aalst 00] W. M. P. Van Der Aalst, Loosely Coupled Interorganizational Workflows : modeling and analyzing workflows crossing organizational boundaries. Journal of Information and Management Vol. 37, n° 2, Pp. 67-75. 2000.

[van Ommering& 00] R. van Ommering, F. van der Linden, J. Kramer, J. Magee, The koala component model for consumer electronics software. Computer 33, 3, 78–85. 2000.

[Vestal& 93] S. Vestal and P. Binns, "Scheduling and communication in MetaH," in Proc. Symp. Real-Time Systems, pp. 194–200. 1993

[Wang& 08] Z.H. Wang, C.J. Guo, B. Gao, W. Sun, Z. Zhang, and W.H. An. A study and performance evaluation of the multi-tenant data tier design patterns for service oriented computing. In e-Business Engineering, 2008. ICEBE'08. IEEE International Conference on, pages 94–101. IEEE, 2008.

[Wermelinger& 02] M. Wermelinger, J. L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, Sci. Comput. Program. 44 (2) 133–155, 2002.

[Weissman& 09] C.D. Weissman and S. Bobrowski. The design of the force.com multitenant internet application development platform. In Proceedings of the 35th SIGMOD international conference on Management of data, pages 889–896. ACM, 2009.

[Wilson& 13] Wilson, A., Baptista, J., Galliers, R.D.,. Performing strategy: aligning processes in strategic IT. In: 34th International Conference on Information Systems Milan, Italy: AIS. 2013.

[Wu& 15] S. P. J. Wu, D.W. Straub, & T.P. Liang. How information technology governance mechanisms and strategic alignment influence organizational performance: Insights from a matched survey of business and IT managers. Mis Quarterly, 39(2), 497-518. 2015.

[Yeow& 18] Yeow, A., Soh, C., & Hansen, R. Aligning with new digital strategy: A dynamic capabilities approach. The Journal of Strategic Information Systems, 27(1), 43-58. 2018.

[Zhang& 10] Q. Zhang, L. Cheng, R. Boutaba. Cloud computing : state-of-the-art and research challenges. Journal of Internet Services and Applications, 1(1) :7–18, 2010.

Annexes

LISTE DES FIGURES

Figure 1: Trame générale des travaux avec publications scientifiques. 6
Figure 2 : Catégories d'architectures logicielles. 17
Figure 3 : Organisation schématique du document. 19
Figure 4 : architectures en couches en génie logiciel (Sysco Labs 2014). 22
Figure 5 : Le cœur de la description de l'architecture selon ISO/IEC/IEEE 42010......................... 26
Figure 6 : Assemblage d'éléments architecturaux pouvant être décrit à l'aide d'un ADL. 32
Figure 7 : Illustration des niveaux d'abstractions de description d'architectures logicielles. 33
Figure 8 : Fiche synthétique de SAEV. 35
Figure 9 : Méta-modèle de SAEV. 37
Figure 10 : Processus opératoire de SAEV. 40
Figure 11: Illustration d'une propriété sémantique. 41
Figure 12 : Fiche synthétique de SAEM. 45
Figure 13 : Méta-modèle de SAEM. 47
Figure 14 : Nom, entête et compétence d'un style d'évolution. 48
Figure 15: concepts du style d'évolution en utilisant le modèle en Y. 50
Figure 16 : Organisation de styles d'évolution. 50
Figure 17 : Les bibliothèques pour les styles d'évolution. 51
Figure 18 : Filtrage par points de vue sur la bibliothèque de niveau E1. 52
Figure 19 : Fiche synthétique. 55
Figure 20 : Modèle du code source. 58
Figure 21 : (a) Eléments de contours – (b) Modèle de correspondance. 59
Figure 22 : Guides de l'extraction. 59
Figure 23 : Modèle de mesure de caractéristiques dans la norme ISO-9126 et dans notre approche. 60
Figure 24 : Notre modèle de mesure de la validité sémantique des composants. 61
Figure 25 : Fiche synthétique. 66
Figure 26: Parallèle entre l'instanciation COSA et Java. 69
Figure 27 : Le patron d'évolution Client-Serveur : application du style architectural C&S............ 70

Figure 28 : Graphe type sous AGG du diagramme de composants d'UML 2. 71
Figure 29 : Contrainte atomique. 71
Figure 30 : Trois de règles de transformations pour déplacer un composant au sein d'un serveur. . 72
Figure 31 : Analyse par paire critique des règles de transformation. 72
Figure 32 : Orchestration et chorégraphie de services [Pahl& 06]. 77
Figure 33 : Les couches d'une SOA [Arsanjani& 07]. 78
Figure 34 : Organisation de l'architecture orientée services. 79
Figure 35 : Aperçu général du Cloud. 79
Figure 36 : Les modèles de fourniture du Cloud ou le modèle SPI. 81
Figure 37 : Les niveaux de maturité du SaaS. 82
Figure 38 : Fiche synthétique. 85
Figure 39 : Schéma global de notre approche. 86
Figure 40 : Méta-modèle de Workflow à base d'activités. 88
Figure 41 : Méta-modèle de définition de processus WF selon une approche SOA. 89
Figure 42 : Démarche générale de restructuration et d'interconnexion de processus. 90
Figure 43 : Méta-modèle d'un Patron de Coopération à Base de Services (PCBS). 91
Figure 44 : Illustration des PCBS : patron « Partage de Charge » - PCBS1 92
Figure 45 : Fiche synthétique. 96
Figure 46 : Un style architectural pour les applications SaaS mutualisées [Koziolek 10]. 98
Figure 47 : La variabilité dans les LDP Vs. la variabilité dans les applications SaaS mutualisées .. 99
Figure 48 : Notre méta-modèle de variabilité. 101
Figure 49 : Exemple d'instanciation du méta-modèle. 102
Figure 50 : Architecture de VaaS. 103
Figure 54 : Fiche synthétique. 106
Figure 55 : Détermination de la granularité des services en fonction des coûts. 108
Figure 56 : Diagramme à quatre niveaux. 111
Figure 57 : les 4 niveaux d'un système d'information. 122

LISTE DES TABLEAUX

Tableau 1 : Notation et illustration d'une règle. 38

Tableau 2 : Une règle d'évolution SAEV. 38

Tableau 3 : Eléments relationnels et axiomes mathématiques des relations en termes de réflexivité (R), transitivité (T) et symétrie (S). 49

Tableau 4: Distribution des opérations sur les éléments COSA 68