Utilisation de la modélisation à l'exécution : objectif, challenges et bénéfices

Olivier Barais

▶ **To cite this version:**

Olivier Barais. Utilisation de la modélisation à l'exécution : objectif, challenges et bénéfices. Informatique et langage [cs.CL]. Université de Rennes 1, 2014. tel-01096952

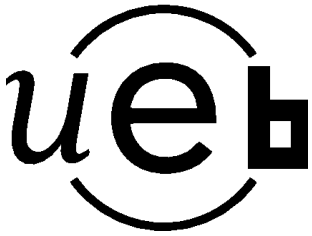
HAL Id: tel-01096952
https://inria.hal.science/tel-01096952

Submitted on 18 Dec 2014

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.

ANNÉE 2014



HABILITATION À DIRIGER DES RECHERCHES
UNIVERSITÉ DE RENNES 1

sous le sceau de l'Université Européenne de Bretagne

Mention : Informatique

École doctorale Matisse

présentée par

Olivier BARAIS

préparée à l'unité de recherche IRISA – UMR6074
Institut de Recherche en Informatique et Système Aléatoires / ISTIC

Utilisation de la modélisation à l'exécution : objectif, challenges et bénéfices.

HDR soutenue à Rennes
le 8 décembre 2014

devant le jury composé de :

Patrick HEYMANS
Professeur à l'Université de Namur / *Rapporteur*

Philippe LALANDA
Professeur à l'Université Joseph Fourier / *Rapporteur*

Michel RIVEILL
Professeur à l'Université de Nice (Ecole Polytechnique) / *Rapporteur*

Examinateur JEAN-LOUIS PAZAT
Professeur à l'INSA de Rennes / *Examinateur*

Examinateur JEAN-MARC JÉZÉQUEL
Professeur à l'Université de de Rennes 1 / *Examinateur*

Examinateur BENOIT BAUDRY
Chargé de Recherche à INRIA Rennes / *Examinateur*

Remerciements

Je remercie Patrick Heymans, Professeur à l'Université de Namur, Philippe Lalanda, Professeur à l'Université Joseph Fourier, et Michel Riveill, Professeur à l'Université de Nice (Ecole Polytechnique), d'avoir bien voulu accepter la charge de rapporteur. Je remercie Jean-Louis Pazat, Professeur à l'INSA de Rennes d'avoir bien voulu juger ce travail.

Il est pour moi évident que le travail d'équipe prime et si j'arrive à mener conjointement l'ensemble des facettes de mon métier avec plaisir c'est que je ne travaille pas seul. Nombreux sont ceux qui peuvent en témoigner et à qui je dois beaucoup :

— Ma directrice de thèse, Laurence Duchien, qui m'a mis le pied à l'étrier du monde de la recherche et qui reste toujours disponible et de très bon conseil.

— Jean-Marc Jézéquel qui m'a accompagné comme un mentor en début de carrière et avec qui je garde un plaisir immense à travailler tant d'un point de vue enseignement, recherche ou dans les missions plus administratives de l'université.

— Les membres de mon équipe de recherche *Triskell* maintenant *DiverSE* avec qui j'ai eu la chance de travailler dans une ambiance de travail toujours excellente. A leur contact, j'ai progressé et appris tous les jours. La qualité de l'esprit d'équipe qu'a su insuffler Jean-Marc Jézéquel et qu'a su entretenir Benoit Baudry par la suite m'ont toujours beaucoup apporté.

— Les doctorants avec qui j'ai eu la chance de mener des recherches : Brice, Gégory, Mickaël, Erwan, François, Inti, Emmanuelle, Paul, Bosco, Julien, Mohammed, Thomas, Édouard. J'ai appris énormément à leur contact et je garde un vrai plaisir à travailler avec eux.

— Jérôme Le Noir et son équipe de Thales Research & Technology avec qui j'ai eu la chance de collaborer depuis 2008 dans des conditions chaleureuses fondées sur une confiance réciproque.

— Mes collègues de l'UFR ISTIC ou de l'ESIR. L'organisation des formations, les réunions pédagogiques, la préparation des habilitations de formation ont été et seront toujours des moments passionnants pour lesquels ce métier reste assez unique.

— Mes anciens collègues lillois que j'ai toujours grand plaisir à retrouver et avec qui j'échange régulièrement.

— Mes étudiants de Master avec qui le contact a toujours été plaisant. Ils m'ont apporté beaucoup par leur curiosité et leur questionnement.

À mon épouse, Claire, et nos trois loulous, Luc, Marion et Marceau, qui tout en me soutenant dans mon activité professionnelle, réussissent dans le même temps à me prévenir d'un engloutissement complet dans cette activité chronophage qu'est l'informatique et tout particulièrement l'enseignement et la recherche dans ce domaine.

À mes parents pour leur soutien sans faille depuis 34 ans, soutien et abnégation allant jusqu'à une relecture de ce mémoire pour tenter de le rendre intelligible.

Avant Propos

Ce manuscrit présente une synthèse de mes travaux de recherche dans le domaine de la modélisation logicielle et son utilisation à l'exécution en vue de présenter mon habilitation à diriger des recherches. Il n'y a pas vraiment de structure type concernant ce genre de document, chaque université ayant plus ou moins sa propre politique. J'ai essayé de rédiger ce document dans l'esprit du texte officiel concernant l'habilitation à diriger des recherches, qui précise par l'arrêté du 23 novembre 1988 (modifié en 1992, 1995 et 2002) : « *L'habilitation à diriger des recherches sanctionne la reconnaissance du haut niveau scientifique du candidat, du caractère original de sa démarche dans un domaine de la science, de son aptitude à maîtriser une stratégie de recherche dans un domaine scientifique ou technologique suffisamment large et de sa capacité à encadrer de jeunes chercheurs.* »

Il est important de noter que l'habilitation à diriger des recherches « *de par sa conception, n'est pas et ne doit en aucun cas être considérée comme un second doctorat, de niveau supérieur, comme l'était auparavant le doctorat d'État par rapport au doctorat de troisième cycle.* » (circulaire no 89-004 du 5 janvier 1989). Dans l'esprit de cette circulaire, je présente dans ce manuscrit une synthèse de mon activité scientifique dans le but de faire apparaître mon expérience dans l'animation d'une recherche de haut niveau. Associé à ce manuscrit, je joins un certain nombre d'articles scientifiques permettant de juger aussi les contributions obtenues. Dans le cadre de cet esprit de synthèse, j'ai pris deux libertés, une sur le style, une sur le fond. Sur le style, certaines parties peuvent utiliser le passé, forme qui est généralement prohibée dans la littérature scientifique. Sur le fond, je ne suis pas revenu en détail sur une comparaison de ces travaux face à l'état de l'art. J'ai jugé que cette comparaison était présente dans les articles scientifiques joints en annexe de ce document. Malgré ces deux points, j'espère que ce document ne donnera pas l'impression d'établir un catalogue de résultats. J'ai en effet essayé tout au long de ce manuscrit de montrer les liens entre mes différents travaux de recherche et mon cheminement.

Table des matières

Table des matières **2**

1 Introduction **7**
1.1 Domaine de recherche . . . 7
1.2 Objectifs et verrous scientifiques . . . 8
1.2.1 Contexte . . . 8
1.2.2 Challenges Liés à l'Ingénierie des Systèmes Adaptatifs Distribués et Hétérogènes . . . 10
1.3 Mon parcours . . . 12
1.4 Ma démarche scientifique . . . 13
1.5 Résumé des travaux présentés et organisation du mémoire . . . 13

I Vers une convergence de l'espace de conception et de l'espace d'exécution **15**

2 Principe du Models@runtime **19**
2.1 Des langages de configuration à la notion de modélisation à l'exécution . . . 19
2.1.1 Contexte : des systèmes reconfigurables de plus en plus complexes . . . 19
2.1.2 Architecture et langages de configuration . . . 21
2.2 Models@Runtime . . . 22
2.2.1 Modélisation des Systèmes Adaptatifs . . . 23
2.2.2 Modélisation par Aspects pour la Dérivation de Configurations . . . 23
2.2.3 (Dé)coupler le modèle de réflexion de la réalité . . . 23
2.3 Validation . . . 25

3 Améliorations des techniques de modélisation pour une utilisation à l'exécution **27**
3.1 Vers une diminution du couplage par rapport au méta-modèle . . . 29
3.1.1 Contexte et problématique . . . 29
3.1.2 Contribution . . . 29
3.1.3 Validation . . . 30
3.2 Composition logicielle . . . 31
3.2.1 Contexte et problématique . . . 31
3.2.2 Contribution . . . 31
3.2.3 Validation . . . 32
3.3 Gestion de la variabilité . . . 33
3.3.1 Contexte et problématique . . . 33
3.3.2 Contribution . . . 33

3.3.3 Validation . . . 35
3.4 Un cadre de modélisation pour son utilisation à l'exécution . . . 37
3.4.1 Contexte et problématique . . . 37
3.4.2 Contributions . . . 38
3.4.2.1 Élicitation des exigences pour un cadre de modélisation utilisable à l'exécution . . . 38
3.4.2.2 EMF : avantages et inconvénients . . . 39
3.4.2.3 KMF : un cadre de modélisation pour les modèles à l'exécution . 39
3.4.3 Validation . . . 40

4 Modèle à l'exécution pour la gestion de systèmes adaptatifs hétérogènes et distribués 43
4.1 Contexte et problématique . . . 43
4.2 Contributions . . . 44
4.2.1 Les caractéristiques de Kevoree . . . 44
4.2.1.1 Séparation entre composants métier et interaction (communication) . . . 45
4.2.1.2 Gestion de la distribution . . . 45
4.2.1.3 Désynchronisation entre le modèle réflexif et le système correspondant . . . 45
4.2.1.4 Dissémination des adaptations . . . 45
4.2.1.5 Hétérogénéité des systèmes d'exécution . . . 45
4.2.2 Quelles abstractions ? . . . 46
4.2.2.1 Paradigmes de modélisation . . . 46
4.2.2.2 Sémantique du modèle de configuration . . . 49
4.2.3 Support de l'hétérogénéité . . . 51
4.2.4 Extensibilité de Kevoree . . . 51
4.2.4.1 Délégation de l'exécution de l'adaptation . . . 52
4.2.4.2 Délégation de la planification de l'adaptation . . . 52

II Expérimentations dans le cadre de ces recherches 55

5 Models@Runtime en environnement mobile 59
5.1 Exemple de mise en œuvre d'un nouveau type de groupe : Gossip . . . 61
5.1.1 Objectifs et spécificité d'une dissémination selon un modèle pair à pair . . 61
5.1.2 Une architecture moyenne calculée comme une agrégation épidémique *gossip* 61
5.1.3 Principe de combinaison des horloges vectorielles ainsi qu'une propagation *gossip* . . . 62
5.1.4 Protocole *gossip* pour dissémination de *Model@Runtime* . . . 62
5.1.4.1 Algorithme principal (voir partie algorithme 3) . . . 63
5.1.4.2 Fonction *SelectPeer* (voir Algorithme 4) . . . 64
5.1.5 Propriétés attendues . . . 65
5.1.5.1 Propriétés de convergence . . . 66
5.1.5.2 Complexité temporelle . . . 66
5.1.5.3 Propriétés de résilience à l'intermittence des connexions . . . 66
5.1.5.4 Complexité en nombre de messages . . . 67
5.1.6 *Slicing* de modèle à l'image du *peer sampling* . . . 67
5.2 Validation expérimentale sur *cluster* de simulation . . . 68
5.3 Protocole expérimental commun . . . 68

5.3.1 Modèle de topologie initiale . . . 69
5.3.2 Horloge de temps absolu pour la collecte des traces d'exécution . . . 69
5.3.3 Mode de communication . . . 69
5.4 Expérimentation 1 : Délai de propagation vis-à-vis de l'usage du réseau de communication . . . 70
5.4.1 Protocole expérimental . . . 70
5.4.2 Limites de validité expérimentale . . . 70
5.4.3 Analyse des résultats expérimentaux . . . 71
5.5 Expérimentation 2 : Impact des erreurs de communication sur les délais de propagation . . . 73
5.5.1 Protocole expérimental . . . 73
5.5.2 Limites de validité expérimentale . . . 74
5.5.3 Analyse des résultats expérimentaux . . . 74
5.6 Expérimentation 3 : Réconciliation de modèle et reconfigurations concurrentes . . . 74
5.6.1 Protocole expérimental . . . 75
5.6.2 Limites de validité expérimentale . . . 75
5.6.3 Analyse des résultats expérimentaux . . . 75
5.7 Conclusion sur l'usage des groupes pour la convergence . . . 76

6 Models@runtime pour le cloud computing 79
6.1 Expérimentation 4 : Est-ce extensible et générique ? . . . 81
6.1.1 Protocole expérimental . . . 81
6.1.2 Implémentation du cas d'étude . . . 82
6.1.2.1 Infrastructure d'espace utilisateur . . . 82
6.1.2.2 Infrastructure de container systèmes . . . 83
6.1.2.3 Proxy pour infrastructure EC2 . . . 84
6.1.3 Évaluation . . . 85
6.2 Expérimentation 5 : Est-ce utilisable pour le pilotage infrastructure de Clouds ? . . . 86
6.2.1 Protocole expérimental . . . 87
6.2.2 Implémentation du cas d'étude . . . 88
6.2.2.1 Plate-forme de déploiement de tests unitaires . . . 90
6.2.2.2 Résultat sur un projet concret : Apache Camel . . . 93
6.2.3 Évaluation . . . 95
6.2.3.1 Impact sur le déploiement . . . 95
6.2.3.2 Impact sur l'utilisation mémoire . . . 96
6.2.3.3 Complexité de l'implémentation de nouveaux types et gestionnaire 96
6.3 Expérimentation 6 : Est-ce utilisable pour de l'adaptation multi-niveaux ? . . . 97
6.3.1 Protocole expérimental . . . 97
6.3.2 Mise en œuvre du cas d'étude . . . 97
6.3.3 Définition du serveur web distribué . . . 98
6.3.4 Évaluation . . . 99
6.4 Synthèse . . . 101

7 Models@runtime pour les objets connectés 103
7.1 Besoins spécifiques des systèmes adaptatifs contraints . . . 104
7.2 Capacité des micro-contrôleurs vis-à-vis des niveaux d'adaptation Kevoree . . . 105
7.3 Implantation d'un nœud Arduino Kevoree . . . 106
7.4 Validation expérimentale sur micro-contrôleur . . . 107
7.4.1 Axes d'évaluation . . . 107
7.4.2 Protocole expérimental général . . . 107

7.5 Expérimentation 7 : Downtime . . . 108
7.5.1 Configuration expérimentale . . . 108
7.5.2 Limites de validité expérimentale . . . 109
7.5.3 Resultats et analyse expérimentale . . . 110
7.5.4 Extension expérimentale pour connaitre l'impact du type de mémoire. . . . 111
7.6 Expérimentation 8 : combien d'instances déployables en mémoire volatile ? . . . 112
7.6.1 Protocole expérimental . . . 112
7.6.2 Limites de validité expérimentale . . . 112
7.6.3 Résultats expérimentaux et analyse . . . 112
7.7 Expérimentation 9 : combien de reconfigurations successives ? . . . 113
7.7.1 Limites de validité expérimentale . . . 114
7.7.2 Résultats et analyse . . . 114
7.8 Expérimentation 10 : Délai de redémarrage . . . 114
7.8.1 Limites de validité expérimentale . . . 114
7.8.2 Résultats expérimentaux et analyse . . . 115
7.9 Comparatif vis-à-vis d'un micro-logiciel non généré . . . 116
7.10 Conclusion vis-à-vis des axes d'évaluation . . . 116

8 Models@runtime pour les systèmes de surveillance dynamiques et optimistes 117
8.1 Contexte . . . 118
8.2 Vue générale de l'approche Scapegoat . . . 118
8.2.1 Contrat de qualité de service . . . 119
8.2.2 Un conteneur muni de mécanismes de surveillance dynamique . . . 120
8.2.3 Architecture de Scapegoat . . . 122
8.2.3.1 Stratégie de mise en œuvre : . . . 122
8.2.3.2 Utilisation du modèle à l'exécution pour la construction d'un framework de monitoring efficace . . . 122
8.3 Protocole expérimental commun . . . 123
8.3.1 Cas d'étude . . . 123
8.3.2 Méthodologie de Mesure . . . 124
8.4 Expérimentation 11 : Coût lié à l'instrumentation . . . 124
8.4.1 Protocole expérimental . . . 124
8.4.2 Résultats expérimentaux et analyse . . . 125
8.5 Expérimentation 12 : Comparaison du coût de la surveillance adaptative face à la surveillance pleine . . . 126
8.5.1 Protocole expérimental . . . 126
8.5.2 Résultats expérimentaux et analyse . . . 126
8.6 Expérimentation 13 : Coût lié à la commutation entre modes de surveillance . . . 127
8.6.1 Protocole expérimental . . . 127
8.6.2 Résultats expérimentaux et analyse . . . 128
8.6.2.1 Résultats . . . 128
8.6.2.2 Impact lié à la taille de l'application . . . 128
8.6.2.3 Impact lié à la taille des composants . . . 129
8.7 Discussions . . . 129
8.7.1 Limites de validité expérimentale . . . 129
8.7.2 Travaux en cours . . . 130

9 Kevoree : une synthèse des idées sous la forme d'un projet *open source* 131
9.1 Outillage associé aux modèles de configuration . . . 132
9.1.1 Notation graphique d'architecture . . . 132

9.1.2 KevScript : DSL de manipulation de modèle d'architecture 133
9.1.3 Model2Code et Code2Model 133
9.1.4 Kevoree IDE, environnement de modélisation d'architecture 138
9.2 Gestion de l'hétérogénéité 139
9.2.1 Implantation des nœuds 139
9.2.2 Maturité du projet 141
9.3 Dissémination et complexité d'apprentissage liées au langage de configuration . . 141

III Conclusion et perspectives 143

10 Bilans d'activités 145
10.1 Bilan scientifique 145
10.2 Bilan quantitatif 147
10.2.1 Bilan des publications 147
10.2.2 Bilan en terme de formation 148
10.2.3 Bilan des coopérations industrielles 148

11 Perspectives et Projet de Recherche 149
11.1 Projet collectif : Vers une démarche de conception fondée sur la diversité choisie pour améliorer la stabilité et la sécurité du système 150
11.2 Perspective 1 : Utilisation du models@runtime pour la maîtrise de la diversité . . 151
11.2.1 Models@runtime pour la synthèse d'infrastructure de tests de systèmes distribués 152
11.2.2 Models@runtime pour l'ingénierie des langages 152
11.2.3 Models@runtime pour le web 153
11.3 Perspective 2 : Amélioration des approches de modélisation pour le support de la diversité 154
11.3.1 Opérateurs de composition de modèles adaptés à un ingénieur système pour la gestion explicite de la variabilité 154
11.3.2 Vers un support de la notion de flux de modèles 154
11.4 Perspective 3 : Coopération application/Système pour le support de la diversité . 155
11.5 Vision : Vers un support technique de l'agilité 156

Bibliographie 159

Table des figures 173

IV Sélection d'articles scientifiques 175

V CV détaillé 287

Chapitre 1

Introduction

Le premier chapitre de ce document de synthèse présente le cadre de ma recherche, à savoir : le domaine de recherche, les objectifs visés et les verrous scientifiques adressés, mon parcours et l'esprit de ma démarche scientifique.

Sommaire

1.1 Domaine de recherche **7**
1.2 Objectifs et verrous scientifiques **8**
1.2.1 Contexte 8
1.2.2 Challenges Liés à l'Ingénierie des Systèmes Adaptatifs Distribués et Hétérogènes 10
1.3 Mon parcours **12**
1.4 Ma démarche scientifique **13**
1.5 Résumé des travaux présentés et organisation du mémoire **13**

1.1 Domaine de recherche

L'ensemble de ma recherche se place dans le domaine dit du génie logiciel. Le génie logiciel est défini comme l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser le développement du logiciel et son suivi. Cette discipline, née de la première crise du logiciel dans les années 1970 [Dij72], a poursuivi son développement du fait de la complexité des systèmes logiciels en croissance permanente. Elle trouve sa justification dans la difficulté intrinsèque à estimer correctement l'effort requis pour aboutir à un logiciel fiable, *agile*[1] et maintenable. Dans ce domaine, je me suis particulièrement focalisé sur les problématiques de modélisation de tels systèmes et j'ai principalement travaillé sur l'utilisation des techniques de modélisation dans les systèmes à l'exécution. Une manière de définir la modélisation est de la rapprocher de la notion d'abstraction. En effet, la démarche de modélisation correspond à la mise en œuvre de la rationalité cartésienne et de la méthode scientifique. Il s'agit tout à la fois, de se simplifier le travail en éliminant les détails, et d'obtenir un résultat plus net, en se concentrant sur les seuls traits jugés importants.

Associé à ce domaine de la modélisation, la notion de modèle est un concept très largement utilisé en science en général. Un modèle est un outil pour penser. C'est une construction qui

1. se dit d'un logiciel que l'on peut faire évoluer facilement en fonction des évolutions des son contexte d'exécution ou des exigences liées à son utilisation

constitue une réponse provisoire et partielle à un problème scientifique. D'une manière générale en science, le modèle permet de représenter, d'expliquer la réalité et d'établir des prévisions. Généralement, un modèle peut être rendu opérationnel ; il s'exprime alors sous forme d'équations mathématiques qui peuvent être implémentées et permettre de réaliser, par exemple, une application informatique permettant de manipuler ce modèle et de réaliser des simulations. Un modèle peut aussi rester conceptuel et prendre la forme de maquettes plus ou moins complexes, de schémas ou de toute autre représentation. De manière général en sciences, les modèles permettent d'abstraire une partie de la réalité, et fournissent une réponse qu'il faudra, dans une démarche scientifique, confronter aux réalités du terrain ou aux résultats expérimentaux. Le modèle représente une réalité, il ne constitue pas cette réalité. Au contraire en informatique, les modèles peuvent constituer la réalité et ne peuvent pas toujours être confrontés au terrain, ils ne sont pas régis par une loi de la nature qu'ils cherchent à imiter. Cette particularité est très importante car ces modèles qui gardent leur capacité de simulation et de vérification peuvent en informatique devenir et constituer la réalité.

1.2 Objectifs et verrous scientifiques

Le développement logiciel « *traditionnel* », généralement fondé sur l'hypothèse d'un monde clos définissant une frontière connue et stable entre le système et son environnement n'est plus tenable. Par opposition, la notion de système dit ouvert et éternel s'est imposée à la plupart des systèmes informatiques. Ces systèmes logiciels se caractérisent par leur besoin d'offrir des capacités d'adaptation qui leur permettent de réagir aux changements de leur environnement de manière continue et sans interruption de service.

Partageant cette vision, un des challenges important qui motive et sous-tend mon activité de recherche est d'identifier et de supprimer progressivement les limites liées à l'hypothèse du monde clos. En partant de cette hypothèse dit de monde ouvert, cette habilitation expose les bénéfices engendrés par l'effacement de la frontière entre la phase de conception et la phase d'exécution du logiciel en proposant l'utilisation des travaux liés à la modélisation non plus uniquement lors de la phase de conception du système, mais aussi au cours de l'exécution des systèmes dits ouverts.

1.2.1 Contexte

L'ensemble de mes recherches s'est placé dans ce contexte de systèmes ouverts en s'intéressant particulièrement à la classe des systèmes logiciels qui sont à la fois :

- **hétérogènes**. Système logiciel s'exécutant sur différents supports d'exécution matériel ou logiciel.
- **distribués**. Système logiciel exécutant ses traitements sur plusieurs unités de calcul (processeur, micro-contrôleur, Machines Virtuelles, Conteneurs, ...).
- **adaptables**. Système logiciel ayant la capacité à évoluer au cours du temps sans interruption de services.

Nous appellerons dans la suite de ce manuscrit cette classe de système « des Systèmes Adaptatifs Hétérogènes et Distribués ».

Outre les nombreuses discussions avec mes collègues et étudiants, trois approches, constats ou tendances ont fortement influencé ce travail de recherche.

La caractérisation des systèmes dynamiquement adaptable DAS) Une première approche prometteuse consiste à concevoir et à implémenter ces systèmes dits ouverts comme

des systèmes adaptatifs (*DAS, Dynamically Adaptive Systems*), qui peuvent s'adapter selon leurs contextes d'exécution, et évoluer selon les exigences utilisateur. Il y a plus d'une décennie, Peyman Oreizy et al. [OGT+99] ont défini l'évolution comme "l'application cohérente de changements au cours du temps", et l'adaptation comme "le cycle de monitoring du contexte, de planification et de déploiement en réponse aux changements de contexte". Cette frontière entre évolution logicielle et adaptation dynamique a tendance à disparaître dans les systèmes adaptatifs.

Les systèmes adaptatifs ont des natures très différentes

- systèmes embarqués [HGC+06] ou systèmes de systèmes [BS06, Got08],
- systèmes purement auto-adaptatifs ou systèmes dont l'adaptation est choisie par un être humain

Si l'on cherche à modéliser la logique d'adaptation et quelque soit son type, l'exécution d'un DAS peut être abstrait comme une machine à états [BBFS08, ZC06], où :

- **les états** représentent les différentes configurations (ou les modes) possibles du système adaptatif. Une configuration peut être vue comme programme "normal", qui fournit des services, manipule des données, exécute des algorithmes, etc.
- **Les transitions** représentent toutes les différentes migrations possibles d'une configuration vers une autre. Ces transitions sont associées à des prédicats sur le contexte et/ou des préférences utilisateur, qui indiquent quand le système adaptatif doit migrer d'une configuration à une autre.

Fondamentalement, cette machine à états décrit le cycle d'adaptation du DAS. L'évolution d'un DAS consiste conceptuellement à mettre à jour cette machine à états, en ajoutant et/ou enlevant des états (configurations) et/ou des transitions (reconfigurations).

L'énumération de toutes les configurations possibles et des chemins de migration reste possible pour de petits systèmes adaptatifs. Ce design en extension permet de simuler et de valider toutes les configurations et reconfigurations possibles, au moment du design [ZC06], et de générer l'intégralité du code lié à la logique d'adaptation du DAS. Cependant, dans le cas de systèmes adaptatifs plus complexes, le nombre d'états et de transitions à spécifier explose rapidement [MBJ+09, FS09] : le nombre des configurations explose d'une manière combinatoire par rapport aux nombres de *features* dynamiques que le système propose, et le nombre de transitions est quadratique par rapport au nombre de configurations. Concevoir et mettre en application la machine à états dirigeant l'exécution d'un système adaptatif complexe (avec des millions ou même des milliards de configurations possibles) est une tache difficile et particulièrement propice aux erreurs.

L'architecture logicielle à base de composants Une deuxième approche complémentaire à la notion de DAS pour maîtriser la complexité et offrir un support de la variabilité à l'exécution est issue du domaine de l'architecture logicielle. *L'architecture logicielle* fournit une abstraction et une approche modulaire dans la conception de système en explicitant la notion de composant logiciel, de connecteur et de configuration. L'idée de composant logiciel date déjà d'un certain nombre d'années. On trouve une des premières utilisations explicites de ce terme en 1968 [McI68]. Cependant, il a fallu attendre les années 90 et la mise en évidence des lacunes du paradigme objet dans le domaine de la réutilisation pour voir apparaître un regain d'intérêt autour de la notion de composant logiciel comme unité de réutilisation de modules logiciels. Pour contrer le problème inhérent à la programmation par objets, identifié sous le nom de *Fragile Base Class* [Szy96, MS98], l'approche par composants propose une meilleure séparation entre la partie réalisation d'un module logiciel et son interface, c'est-à-dire la description de

services qu'il fournit et qu'il requiert. Chaque composant est une boîte noire, indivisible, composable, déployable et identifiable par les services qu'elle offre et qu'elle requiert. De plus, de nombreuses approches ont montré le bénéfice de son utilisation dans le cadre du développement et de l'administration des systèmes distribués [IBRZ00, HRPL+95, MT00].

Le découplage entre la partie réalisation d'un composant et son interface avec l'environnement met en avant deux informations : une description claire de l'interface des composants et une description de l'assemblage des composants, c'est-à-dire leurs interactions. Ces deux informations au cœur de l'approche par composants constituent les éléments de base de ce que l'on nomme l'*architecture logicielle*. L'utilisation d'une description d'architecture logicielle apporte quatre avantages dans un projet informatique [GS93]. Elle facilite la compréhension de la structure d'un système. Elle permet son analyse. En tant que cadre pour la construction de l'application, elle permet la génération de code et l'identification des opportunités de réutilisation. Enfin, elle définit les invariants du système et sert donc de pivot pour son évolution.

De l'agilité à la livraison de logiciels en continu. Le troisième constat qui a fortement influencé mes travaux de recherche est lié à l'évolution des méthodologies de développement qui ont également connu des modifications profondes pour répondre au besoin d'évolution continue de ces logiciels. Si le cycle en V était préconisé dans les années 80, à savoir conception et spécification initiale puis génération ou implémentation de code, les méthodes modernes préconisent l'hyper agilité[Sto09]. En effet dans une étude publiée en 2011 [Ver10] sur l'adoption industrielle des méthodes Agiles, *VersionOne* annonce que 80% des sondés déclarent que leur compagnie a adopté un tel processus. Ces nouvelles méthodes cherchent à introduire des cycles plus courts de développement afin de répondre plus rapidement à des évolutions de spécification et surtout à obtenir très tôt des retours utilisateurs. Cette hyper agilité s'accompagne d'une nouvelle approche pour le test et l'intégration des nouveaux développements : l'intégration continue [FF06]. Dans cette approche un système de tests est remis à jour avec les derniers artefacts de développement en continu, le plus souvent sans réinstallation particulière de logiciels. Cette méthodologie étendue jusqu'à la notion de livraison continue (*continuous delivery*) [HF10] et que l'on retrouve aussi dans le domaine de l'informatique de gestion a tendance à apporter de plus en plus les contraintes définies pour les DAS à tout système informatique. Cette tendance étaye la thèse du besoin de rapprochement entre les abstractions utilisées pour les systèmes adaptables, distribués et hétérogènes et les systèmes informatiques dits non critiques. Le cycle de développement et de livraison se fait ainsi maintenant de manière continue sur des systèmes critiques ou non.

1.2.2 Challenges Liés à l'Ingénierie des Systèmes Adaptatifs Distribués et Hétérogènes

Betty Cheng *et al.* ont identifié plusieurs défis liés aux DAS [CLG+09], spécifiques à différentes activités : modélisation des différentes dimensions d'un DAS, expression des besoins, ingénierie (design, architecture, vérification, validation, etc) et assurance logicielle. Ces défis peuvent être complétés par l'analyse de Baresi *et al.* sur les challenges pour la construction de systèmes ouverts. La plupart de ces défis peuvent se résumer à trouver le niveau juste de l'abstraction :

— assez abstrait pour pouvoir raisonner efficacement et exécuter des activités de validation, sans devoir considérer tous les détails de la réalité,
— et suffisamment détaillé pour établir le lien (dans les 2 directions) entre l'abstraction et la réalité, c'est-à-dire pour établir un lien causal entre un modèle et le système en cours d'exécution.

L'abstraction est l'une des clés pour maîtriser la complexité des logiciels. La clé pour maîtriser les systèmes dynamiquement adaptatifs [MBJ+09] est de réduire le nombre d'artefacts qu'un concepteur doit spécifier pour décrire et exécuter un tel système, et d'élever le niveau d'abstraction de ces artefacts. Selon Jeff Rothenberg [RWLN89],

> *Modéliser, au sens large, est l'utilisation rentable d'une chose au lieu d'une autre pour un certain but cognitif. Cela permet d'employer quelque chose qui est plus simple, plus sûr ou meilleur marché que la réalité, pour un but donné. Un modèle représente la réalité pour un but donné ; c'est une abstraction de la réalité dans le sens qu'il ne peut pas représenter tous les aspects de la réalité. Cela permet d'envisager le monde d'une façon simplifiée, en évitant la complexité, le danger et l'irrévocabilité de la réalité.*

C'est dans cette optique et pour répondre à la complexité d'assemblage des systèmes modernes qu'a émergé la mouvance de méthodes et d'outils autour de l'ingénierie dirigée par les modèles (IDM). L'IDM appliquée à la construction de logiciels vise donc à fournir des possibilités de points de vue différents et simplifiés d'un système informatique. Ces outils ont été exploités et ont démontré leur intérêt lors de la phase de conception d'un système permettant de vérifier, d'évaluer des propriétés sur ce dernier ou de produire automatiquement une partie de celui-ci.

Dans les DAS complexes, un des aspects importants de la réalité que nous voulons abstraire est entre autres le système lui-même. Ceci soulève les questions de recherche (RQ) suivantes :

1. **RQ1.** Que proposer dans cette abstraction afin que l'architecte d'un système adaptatif distribué et hétérogène puisse spécifier, déployer, reconfigurer et exécuter un tel système ? Quels concepts permettent de capturer les aspects essentiels d'un système distribué dynamiquement adaptable en cours d'exécution ? Une telle couche d'abstraction doit expliciter et non masquer les problèmes du domaine d'étude. Ceci est nécessairement vrai dans le monde du distribué en accord avec le constat de Guerraoui et al [GF99] qui parlent de : « mythe de la distribution transparente »mais aussi dans un monde hétérogène ou l'uniformisation par l'intersection des propriétés des constituants du système se retrouve toujours être inutile sur des cas réels ?
2. **RQ2.** Comment établir un lien entre la configuration (architecture) correspondant à un ensemble de fonctionnalités, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation ?
3. **RQ3.** Comment faire migrer un DAS de sa configuration courante vers une nouvelle configuration sans devoir écrire à la main tous les scripts de bas niveau et spécifiques à une plateforme d'exécution donnée ?
4. **RQ4.** Comment maintenir la cohérence de ce modèle représentant l'état du système dans un environnement distribué ? Ce point reste un challenge particulièrement dans des environnements mobiles où les communications sont intermittentes ?
5. **RQ5.** Comment valider la pertinence de l'abstraction proposée et sa pertinence pour le domaine visé ?
6. **RQ6.** Comment réutiliser les approches, méthodes et outils habituellement utilisés sur un modèle de conception, à l'exécution ?
7. **RQ7.** Comment adapter les approches, méthodes et outils de modélisation à l'hypothèse du monde ouvert ?

Ces questions amènent à élever le niveau d'abstraction, et à réduire le nombre d'artefacts requis pour spécifier et exécuter des systèmes hétérogènes distribués dynamiquement adaptables, tout en préservant un degré élevé de validation, d'automatisation et d'indépendance par rapport

à des choix technologiques (par exemple, outils utilisés au design, plate-formes d'exécution, système de règles pour diriger l'adaptation dynamique, etc.). Mais ces questions amènent aussi à adapter les techniques de modélisation pour la prise en compte de l'hypothèse du monde ouvert et leur utilisation à l'exécution.

1.3 Mon parcours

Les recherches que j'ai effectuées ont débuté à Lille sous la direction de Laurence Duchien de 2002 à 2005 et ont ensuite été menées sous la caution/direction scientifique de Jean-Marc Jézéquel et Benoît Baudry au sein de l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) à Rennes au sein de l'équipe Triskell de 2006 à 2013 puis DiverSE depuis début 2014. Durant ces années le fil directeur de ma recherche a été l'adaptation des techniques d'ingénierie logicielle à l'hypothèse du monde ouvert et en particulier l'adaptation des techniques de modélisation.

De 2002 à 2005, j'ai travaillé dans le domaine de la modélisation d'architecture logicielle. Ma thèse a proposé un cadre de description d'architectures logicielles complet, permettant de spécifier une architecture logicielle de manière incrémentale en travaillant sur l'analyse d'une description d'architecture et son utilisation pour le prototypage rapide d'une application. Deux axes principaux ont structuré ces travaux. Le premier axe a permis la définition d'un modèle abstrait de description d'architecture logicielle à base de composants. Ce modèle rend possible la validation d'une description d'architecture logicielle et la génération de code vers différentes plates-formes à composants existantes pour un prototypage rapide de l'application. Le deuxième axe a étendu ce modèle afin de favoriser l'intégration de nouvelles préoccupations au sein d'une architecture logicielle. Inscrit dans une démarche de séparation des préoccupations au niveau de la conception de l'application, j'ai proposé dans ma thèse un mécanisme de transformation permettant de tisser de façon fiable ces nouvelles préoccupations au sein d'une architecture logicielle existante. Durant ma thèse j'ai donc eu l'opportunité d'aborder certaines facettes de RQ1-5-7.

Á partir de 2006, j'ai travaillé plus généralement sur les outils de modélisation et leur utilisation à l'exécution. De 2006 à 2009, j'ai cherché dans le cadre de la thèse de Brice Morin à utiliser des concepts de modélisation à l'exécution (*Models@runtime*) dans le cadre de systèmes dynamiquement adaptables centralisés et homogènes. Cette période m'a permis d'étudier les questions de recherches RQ2, RQ3, et RQ6.

Durant la période de 2007 à 2010, j'ai également travaillé à valider cette approche dans un domaine d'application comme celui de la domotique. Parallèlement à ces travaux, j'ai amélioré les approches de modélisation pour les confronter à l'hypothèse du monde ouvert. Dans le cadre des travaux de thèse de Mickaël Clavreul, j'ai contribué à la définition de nouveaux opérateurs de composition permettant d'aborder les challenges liés à RQ3. J'ai aussi participé à la maturation de la notion de type de modèle introduite par Jim Steel et Jean-Marc Jézéquel [SJ07], notion clé pour permettre d'utiliser des approches MDE de construction de langages dans l'hypothèse du monde ouvert (RQ7).

A partir de 2008, autour de RQ1, RQ5, et RQ7, j'ai mené des recherches au travers de différents partenariats industriels liés à l'ingénierie système sur la modélisation de variabilité dans une ingénierie multi-vues. J'ai en particulier non pas contribué à la notion de ligne de produit mais étudié :

1. le lien entre la modélisation de la variabilité et les artefacts de modélisation système associé
2. et les capacités d'opérationalisation de la dérivation d'une ligne de produit et son utilisation à l'exécution.

Enfin à partir de 2009, j'ai travaillé davantage dans le cadre des systèmes adaptables distribués et hétérogènes en abordant en particulier sur RQ1 et RQ4 et en confrontant les résultats à différents domaines d'applications : le *cloud computing*, le Web, le test. Ces travaux m'ont amenés à travailler en particulier sur la question de recherche RQ6.

1.4 Ma démarche scientifique

Si les paragraphes présentés précédemment m'ont permis de présenter mon domaine de recherche, ses motivations, son cadre d'application et ses limites, ce paragraphe vise à présenter ma démarche scientifique.

Premièrement, si naturellement j'aime adopter une démarche rationnelle et constructiviste au travers d'une approche déductive où la vérité émane de constructions logiques et de schémas conceptuels, il est à noter que le domaine du génie logiciel, en particulier quand on se place dans l'hypothèse du monde ouvert ne se valide généralement que de manière expérimentale. C'est pourquoi, depuis ces douze ans de recherche, j'ai toujours eu à cœur d'implanter les idées et modèles proposés afin de permettre une évaluation expérimentale par des étudiants, des collaborations industrielles ou des membres de mon équipe de recherche.

Passionné par les aspects techniques de l'informatique, la curiosité envers la technologique est le deuxième pilier de ma démarche scientifique. Je trouve très important d'évaluer, de *prototyper*, de tester et d'implanter en utilisant les technologies les plus à jour afin de conserver la compréhension des problématiques actuelles du développement logiciel.

Troisièmement, si ce document est, par moment, écrit à la première personne pour un effet de style, il est à noter qu'il résulte d'un travail collectif fortement dopé par les masters et doctorants avec qui j'ai eu la chance de travailler mais aussi les collèges des équipes Triskell, DiverSE, Spirals, Serval, Tram, ThingMLTeam et Myriads. Cette chance offerte par des laboratoires comme l'IRISA ou le LIFL de travailler en équipe permet entres autres de croiser les points de vue, les domaines d'applications et de *cross-fertiliser* les idées des uns et des autres. L'importance du travail en équipe est le troisième pilier de ma démarche scientifique, comme il est nécessaire d'expérimenter, de nombreuses ressources sont nécessaires à l'accomplissement de recherches de bon niveaux.

Finalement, travaillant sur des domaines appliqués, mener une recherche en collaboration avec des industriels me semble le dernier pilier d'une bonne recherche. La confrontation aux problématiques industrielles imposent deux contraintes qui me semblent importantes dans mon domaine de recherche.

1. éviter de construire des modèles sur des hypothèses erronées.
2. être pédagogique sur les réponses que l'on apporte afin d'expliquer en quoi ces dernières peuvent améliorer les pratiques existantes.

La curiosité technique, la validation au travers de l'implantation et l'expérimentation, la *cross-fertilisation* d'idées en équipe, et la confrontation au monde industriel sont les quatre piliers qui sous-tendent mon activité de recherche depuis douze ans.

1.5 Résumé des travaux présentés et organisation du mémoire

Cette habilitation synthétise dans un premier temps les fondations d'une approche permettant l'utilisation de techniques de modélisation à l'exécution, en se concentrant principalement sur le point de vue de l'architecte logiciel en charge du déploiement et de la configuration

logicielle. Nous exposons ensuite les bénéfices attendus en montrant comment des approches avancées de composition logicielle, de vérification ou de gestion de la variabilité peuvent être bénéfiques pour la compréhension et la maîtrise de l'espace de configuration et de reconfiguration d'un système dit ouvert. Nous synthétisons ensuite les principaux challenges liés à l'utilisation de techniques de modélisation à l'exécution en particulier dans le cadre de systèmes distribués et hétérogènes. Enfin, nous validons cette approche en la confrontant à différents domaines d'applications : les environnements mobiles, le monde des objets connectés, le *cloud computing* et la surveillance de consommation de ressources. Pour chacun de ces domaines, nous cherchons à pousser aux limites cette idée d'utilisation de modélisation à l'exécution en regardant sa pertinence pour chaque domaine étudié par rapport à ses propres contraintes.

La première partie de ce manuscrit présente la recherche effectuée pour obtenir une meilleure convergence de l'espace de conception et de l'espace d'exécution. Ce chapitre est présenté comme un résumé commenté de 8 articles scientifiques que je considère comme clé pour présenter le cheminement de mes activités de recherche depuis dix ans. Cette partie est volontairement courte, les articles en question étant publiés et joints en annexe du manuscrit. Dans cette partie, le chapitre 2 présente la contribution liée à la définition de la notion de *models@runtime*. Le chapitre 3 présente les améliorations des techniques de modélisation pour leur utilisation à l'exécution dans l'hypothèse d'un monde ouvert, en insistant en particulier sur quatre propositions dans le domaine de la méta-modélisation, dans le domaine de la gestion de la variabilité, dans le domaine de la composition de modèles, et dans l'amélioration des techniques de modélisation pour leur potentielle utilisation à l'exécution. Le chapitre 4 résume la proposition liée à l'utilisation du *models@runtime* dans un contexte de systèmes adaptatifs hétérogènes et distribués.

La deuxième partie de ce manuscrit revient sur les différents efforts de validation liés à ces recherches en particulier, il revient sur Kevoree , le *framework* de models@runtime pour les systèmes distribués et hétérogènes, présente les expérimentations effectuées pour l'utilisation de Kevoree dans un contexte d'informatique ubiquitaire, dans un contexte de *cloud computing* et dans un contexte lié à l'Internet de Objets. Il présente aussi une expérimentation pour la surveillance dynamique et optimiste de consommation de ressources.

Le chapitre 10 conclut ce mémoire de synthèse au travers de plusieurs points : un résumé des contributions scientifiques proposées, un bilan quantitatif en termes de production scientifique, de formation et de coopérations industrielles. En guise de perspectives, le chapitre 11 présente l'ensemble des axes de recherche en cours. Chacun de ces axes constitue un prolongement naturel des recherches présentées dans ce mémoire : systèmes adaptatifs, langage dédié, ingénierie dirigée par les modèles, gestion de l'hétérogénéité et de la diversité. Comme preuve de l'effort constant pour contribuer aux travaux de la communauté académique ou industrielle, j'insiste en particulier sur les partenariats et contrats qui soutiennent ces axes.

Au sein des annexes de ce manuscrit, j'ai regroupé quelques publications qui servent de support à ce mémoire. Suivent un curriculum vitae et une bibliographie personnelle.

Première partie

Vers une convergence de l'espace de conception et de l'espace d'exécution

Cette partie est un ensemble de résumés d'articles commentés. Son objectif est de présenter le cheminement de mes activités de recherche depuis dix ans en illustrant ce cheminement au travers d'une sélection de huit publications jugées clés dans ce parcours.

Le choix de ces articles permet de montrer un certain nombre de contributions proposées afin d'adresser les questions de recherche $RQ1$, $RQ2$, $RQ3$, $RQ6$ et $RQ7$. Cette collection d'articles introduit une démarche globale pour la modélisation et la gestion à l'exécution de systèmes complexes en particulier dans le cadre de systèmes adaptatifs hétérogènes et distribués centrés autour de quatre points de vue intégrés :

— son architecture, qui décrit la configuration du système courant en termes de concepts architecturaux.
— la variabilité du système, qui décrit les différentes fonctionnalités (*features*) du système, et leurs natures (options, alternatives, etc.)
— la logique d'adaptation du système, qui décrit quand le système doit s'adapter. Cela consiste à définir quelles fonctionnalités (*features*) (du modèle de variabilité) choisir en fonction du contexte courant.
— l' environnement du système, qui décrit les points importants du contexte d'exécution à surveiller.

Le chapitre 2, en résumant [MBNJ09, MBJ+09], présente cette vision générale.

Le chapitre 3 résume et commente quatre contributions [SMM+12, CBJ10, FFBA+14, FNM+12] qui ont permis de rendre cette vision réaliste et maintenable par :

— l'adaptation des techniques de modélisation à l'hypothèse d'un monde plus ouvert dans lequel les différents méta-modèles proposés sont amenés à évoluer et dans lequel de nouveaux points de vue peuvent être intégrés (section 3.1),
— l'intégration de codes patrimoniaux à des approches de modélisation à l'aide de techniques de composition de modèle (section 3.2),
— la création de modèles de variabilité orthogonaux à différents points de vue mais adaptés à l'hypothèse du monde ouvert dans lequel les différents méta-modèles proposés pour chaque point de vue sont amenés à évoluer et dans lequel de nouveaux points de vue peuvent être intégrés (section 3.3),
— l'amélioration des techniques de modélisation pour leur utilisation à l'exécution (section 3.4).

Enfin le chapitre 4 [FDP+12b, FMF+12] détaille le point de vue d'architecture proposé pour la modélisation et l'administration de systèmes adaptatifs hétérogènes et distribués.

Chapitre 2

Principe du Models@runtime [MBNJ09, MBJ+09]

Ce chapitre résume une première contribution autour de l'utilisation de modèles à l'exécution. Il correspond à un résumé court des deux articles suivants [MBNJ09, MBJ+09] publiés dans le cadre de la thèse de Brice Morin. Ces deux articles sont fournis en annexe de ce manuscrit pour étayer cette première proposition qui a fondé le travail sur l'utilisation des modèles à l'exécution.

Sommaire

2.1 Des langages de configuration à la notion de modélisation à l'exécution . . **19**
2.1.1 Contexte : des systèmes reconfigurables de plus en plus complexes . 19
2.1.2 Architecture et langages de configuration 21
2.2 Models@Runtime **22**
2.2.1 Modélisation des Systèmes Adaptatifs 23
2.2.2 Modélisation par Aspects pour la Dérivation de Configurations . . 23
2.2.3 (Dé)coupler le modèle de réflexion de la réalité 23
2.3 Validation **25**

2.1 Des langages de configuration à la notion de modélisation à l'exécution

2.1.1 Contexte : des systèmes reconfigurables de plus en plus complexes

La société d'aujourd'hui dépend de plus en plus des systèmes logiciels [Ben09] déployés dans de grandes compagnies, banques, aéroports, opérateurs de télécommunication, etc. Ces systèmes doivent être disponibles 24H/24 et 7j/7 pour de très longues périodes. Ainsi, le système doit être capable de s'adapter à différents contextes d'exécution, sans interruption (ou très localisé

dans le temps et dans l'espace), et sans intervention humaine. Pour pouvoir s'exécuter pendant une longue période, le système doit être ouvert à l'évolution. Il est en effet impossible de prévoir ce que seront les besoins des utilisateurs dans 10 ans.

Dans ce cadre, une technique souvent utilisée afin de pouvoir anticiper les changements consiste à construire des systèmes de manière modulaire. Ces modules facilement installables et configurables communiquent au travers d'interfaces clairement définies. Cette problématique d'anticipation du changement et de plasticité d'une application a tendance à complexifier fortement la tâche de configuration d'une telle application. Pour illustrer cette complexité de configuration et de spécialisation, nous pouvons prendre deux exemples :

Le premier est la construction d'une application domotique déployée sur l'agglomération de Rennes. Dans ce type d'application, nous montrons dans [MBNJ09] que le nombre de configuration peut vite atteindre une taille très importante. En considérant cinq préoccupations (les protocoles et les types d'équipement utilisés, l'internationalisation, la gestion des droits, et le modèle d'architecture), nous montrons que le nombre de configurations possibles dépasse 15^{13}. Par conséquent, dans l'absolu, le nombre de transitions possibles à définir et à vérifier entre chacune de ces transitions dépasse 225^{26}, amenant une vraie difficulté pour spécifier ces transitions et ces configurations mais aussi les valider.

Le deuxième exemple montre cette même problématique de configuration pour la suite applicative OpenStack [Opeb]. OpenStack est un ensemble de logiciels open source permettant de déployer des infrastructures de Cloud computing (*Infrastructure as a service*). La technologie possède une architecture modulaire composée de plusieurs projets corrélés (Nova, Swift, Glance...) qui permettent de contrôler les différentes ressources des machines virtuelles telles que la puissance de calcul, le stockage ou encore le réseau inhérent au *datacenter* sollicité. Voici la liste des composants intégrés à OpenStack :

- *Compute* : **Nova** (permet de piloter la création de machine virtuelle ou de container)
- *Object Storage* : **Swift** (permet de gérer le stockage d'objet)
- *Image Service* : **Glance** (propose un service d'image prêt à être déployée)
- *Dashboard* : **Horizon** (propose une interface Web de paramétrage et gestion)
- *Identity* : **Keystone** (propose une gestion de l'identité et des droits au sein d'openstack)
- *Network* : **Neutron** (propose une gestion des réseaux à la demande)
- *Storage* : **Cinder** (propose un service de disques persistants pour les machines virtuelles)
- *Orchestration* : **Heat** (propose un service d'orchestration à base de template)
- *Telemetry* : **Ceilometer** (fournit un service de métrologie notamment pour la facturation)
- *Trove* : fournit un service de base de donnée à la demande)

Il existe aussi des composants qui sont dits en incubation car ils ne sont pas encore suffisamment stables pour être intégrés. Cependant, ils sont régulièrement déployés tout de même par certains utilisateurs. On peut citer parmi ces services en incubation :

- **Ironic** : qui offre un service de déploiement sur infrastructure physique,
- **TripleO** (*OpenStack On OpenStack*) : qui propose un service de déploiement de cloud OpenStack grâce à OpenStack,
- **Marconi** : qui propose un service de Middleware à la demande,
- **Sahara** : qui fournit un service d'Hadoop [SKRC10] à la demande.

L'ensemble de ces composants communiquent au travers d'un bus de messages (AMQP pour *Advanced Message Queuing Protocol*[Vin06]) et d'un mécanisme de RPC objet [BN84] (pour *Remote Procedure Call*) construit par dessus. Un extrait d'architecture présenté en figure 2.1 montre la complexité potentielle de telles architectures. Si l'on modélise ensuite la plate-forme d'exécution et les applications afin de pouvoir envisager des adaptations transverses et coordonnées, la complexité globale de tels systèmes se perçoit vite. On peut l'apercevoir à deux

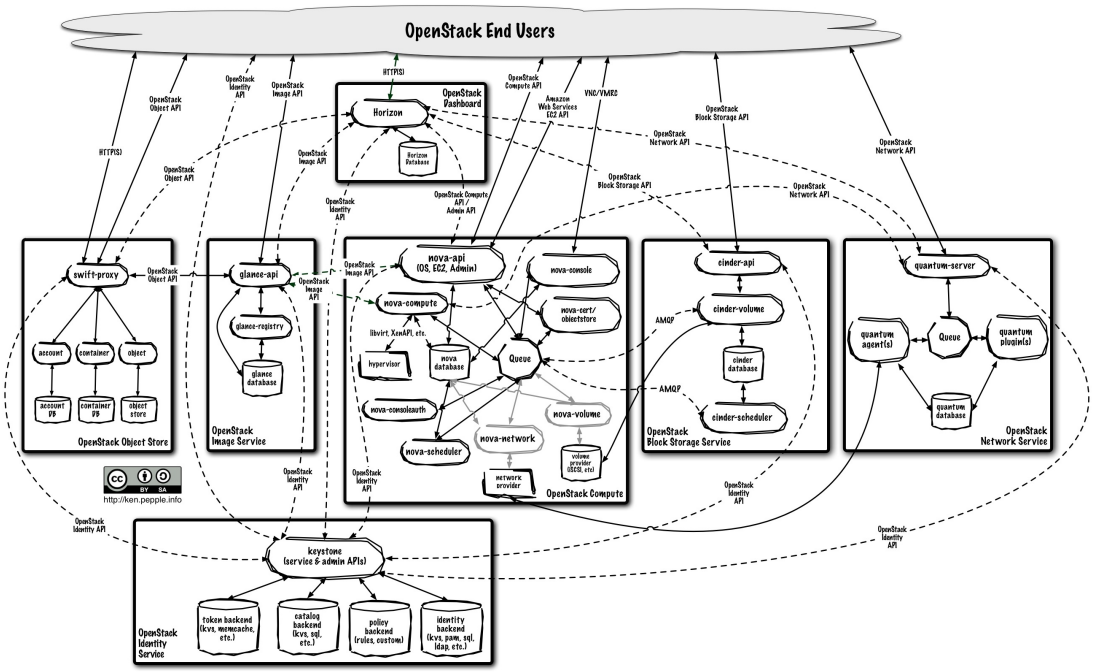


FIGURE 2.1 – Openstack architeture from [Opeb]

niveaux :

- tout d'abord, au travers d'une analyse du projet devstack [1] qui sert uniquement à configurer une plate-forme de préproduction pour les développeurs *OpenStack*. Ce projet contient environ 56668 lignes de code correspondant à : du script Bourne Shell pour 69 fichiers et 5662 lignes, du code Python 245 lignes du code, des gabarits de fichiers de configuration, primitives d'installation, ...pour le reste.
- Ensuite, par construction, le nombre de configuration possible n'est pas quantifiable. De nombreux paramètres n'ont pas de valeur maximum connue (nombre d'instance nova par exemple).

Ces deux exemples montrent le besoin de techniques pour modéliser la configuration permettant une prise en compte des points de variations et rendant possible la reconfiguration dynamique de tels systèmes et la validation des configurations et des reconfigurations.

2.1.2 Architecture et langages de configuration

Parmi l'ensemble des langages de description d'architecture logicielle, de nombreux langages de configuration ont été proposés. Ainsi, la communauté scientifique en architecture logicielle a été particulièrement prolifique à la fin des années 90 [HRPL+95, MT00]. Un langage de configuration favorise la description de la configuration d'une application, c'est-à-dire la description des interactions entre composants. La particularité de ces langages est généralement qu'un composant est une entité instanciable. La description d'un composant au niveau du langage permet de créer de multiples instances d'un composant lors de l'exécution et de configurer les propriétés de ces composants. De nombreux projets ont eu une réelle influence sur la communauté dont

1. http://www.devstack.org

des projets on peut ainsi citer Darwin [KM85] ou même Olan [BDPF00, BBB+98] que l'on peut voir comme un langage qui a indéniablement influencé l'initiative autour de Fractal [BCL+06].

Il est souvent reproché aux langages de description d'architecture leur manque de pénétration dans des produits industriels. Cependant, une des instanciations de ces recherches dans un produit industriel que l'on peut prendre en exemple est le *framework Spring* [2]. Spring est généralement présenté comme un conteneur dit « léger ». La raison de ce nommage vient du fait que Spring par défaut ne prend en charge que la préoccupation lié au cycle de vie des objets, leur création, leur mise en relation d'objets et leur configuration par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets. Par conséquent, ce type de conteneur est non invasif. Il peut piloter la configuration d'objets simples.

Pour notre propos dans ce manuscrit, ce qui nous intéresse dans le succès de Spring est qu'il s'appuie principalement sur l'intégration de trois concepts clés :

— un modèle de configuration abstrait : Cette couche d'abstraction permet d'intégrer d'autres *frameworks* et bibliothèques avec une plus grande facilité.
— la définition d'un cadre de développement fondée autour de l'inversion de contrôle [Fow04]. L'inversion de contrôle est assurée de deux façons différentes : la recherche de dépendances et l'injection de dépendances.
— un ensemble d'opérateurs de composition de haut niveau avec entre autre, le support de la programmation par aspects [KLM+97].

Parmi les points positifs de Spring, il est souvent noté que ce type de cadre de développement permet de maintenir des bonnes pratiques dans des équipes de développement (injection de dépendances, séparation claire entre interfaces et implémentations, programmation par composants, ...). Ceci poussant à conserver une séparation claire entre la problématique de configuration et la problématique de développement facilitant la collaboration entre les développeurs, les testeurs et les personnes en charge du déploiement et de l'administration des applications.

Pour autant, le modèle de configuration de Spring présente plusieurs limites, il ne propose pas de support avancé pour la modélisation de la variabilité au sein du modèle de configuration. Il ne permet pas de modéliser la dynamique de la configuration. Il n'offre pas de mécanisme de composition avancée au niveau du modèle de configuration pour fusionner ou projeter un modèle de configuration ou calculer les différences entre modèles de configuration. Autant de points nécessaires pour permettre de modéliser la complexité de configuration de Systèmes Adaptatifs Hétérogènes et Distribués.

2.2 Models@Runtime

Pour supporter la complexité de configuration de systèmes adaptatifs hétérogènes et distribués, nous avons proposé de tirer parti des dernières avancées en Ingénierie des Modèles (MDE, Model-Driven Engineering [Sch06]) aussi bien pendant la conception (design) qu'à l'exécution (runtime) [BBF09]. Nous proposons d'abstraire les aspects fondamentaux d'un système complexe éternel (sa variabilité (dynamique), son contexte, sa logique d'adaptation et son architecture) à l'aide de méta-modèles/langages dédiés. En opérant à un niveau élevé d'abstraction, il est ainsi possible de raisonner efficacement, en cachant les détails non pertinents, et il devient aussi possible d'automatiser le processus de reconfiguration dynamique. Au travers des deux articles [MBJ+09, MBNJ09] et dans le cadre de la thèse de Brice Morin, nous avons proposé des contributions pour répondre, en particulier, aux questions suivantes :

1. **RQ2.** Comment établir un lien entre la configuration (architecture) correspondant à un

2. http://spring.io

ensemble de fonctionnalités, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation ?

2. **RQ3.** Comment faire migrer un système adaptable de sa configuration courante vers une nouvelle configuration sans devoir écrire à la main tous les scripts de bas niveau et spécifiques à une plateforme d'exécution donnée ?

2.2.1 Modélisation des Systèmes Adaptatifs

Dans le contexte de la thèse de Brice Morin, nous proposons de modéliser un système adaptatif à partir de **quatre points de vue** :

— sa variabilité, qui décrit les différentes fonctionnalités (*features*) du système, et leurs natures (options, alternatives, etc.)
— son environnement, qui décrit les points importants du contexte à surveiller.
— sa logique d'adaptation, qui décrit quand le système doit s'adapter. Cela consiste à définir quelles fonctionnalités (*features*) (du modèle de variabilité) choisir en fonction du contexte courant.
— son architecture, qui décrit la configuration du système courant en termes de concepts architecturaux.

Cette contribution est présentée dans [MBJ+09].

2.2.2 Modélisation par Aspects pour la Dérivation de Configurations

La communauté SPL [BGH+06a, CN01, ZJ06] propose déjà des formalismes et des notations bien établis, tels que des *feature diagrams* [SHT06], pour contrôler la variabilité en fond décrivant une gamme de produits sans devoir énumérer tous les produits individuellement. La communauté SC propose un éventail de techniques différentes mais complémentaires, qui peuvent être utilisées pour dériver des produits depuis un modèle de ligne de produits [KAB07, PKGJ08, MFB+08]. Nous nous appuyons sur cette expérience pour décrire la variabilité de systèmes adaptatifs hétérogènes et distribués, et dériver des configurations depuis un modèle de variabilité. Puisque nous utilisons intensivement des modèles, nous proposons d'utiliser la Modélisation par Aspects (AOM, *Aspect-Oriented Modeling*) afin de raffiner et composer les *features* [PKGJ08, GV08, WJ07]. L'AOM se base sur des approches formelles, telle que la théorie des graphes [BM76], ce qui permet de valider tôt dans le cycle de développement, sans devoir énumérer toutes les configurations possibles (combinaisons de *features*). Alors que la littérature est très prolifique au sujet des approches AOM [BC04, LMV+07, MKBJ08, JWEG07, GV08], peu d'implémentations sont réellement disponibles. Dans la thèse de Brice Morin, nous présentons comment nous étendons les approches *SmartAdapters* et TranSAT [LMV+07, MBJR07, BLLMD06], que nous avons développées. Cependant, notre approche n'est pas spécifique à un tisseur d'aspects. Il est en effet possible d'utiliser d'autres tisseurs, voire même de simples transformations de modèles pour produire des configurations.

Cette contribution est détaillée dans [MBNJ09].

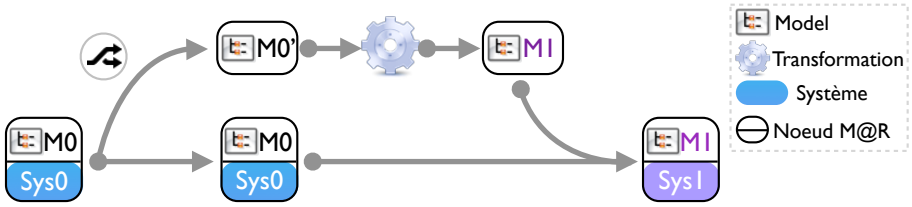
2.2.3 (Dé)coupler le modèle de réflexion de la réalité

De manière schématique nous proposons d'utiliser une démarche dite de modélisation à l'exécution (M@R pour *models@runtime*). Cette approche permet d'extraire et modifier le modèle réflexif du système concret (ex : ajout/suppression d'un composant), puis de détecter et déployer toute modification par différence de versions vers le système en cours d'exécution (ex : déploiement du composant). Inversement, toute modification du système passe par une modification

dans la couche de Model@Runtime (M@R) introduisant alors un lien causal bidirectionnel. Le modèle à l'exécution devient donc une couche de réflexion qui peut être exploitée de manière asynchrone : en d'autres termes, il est possible de modifier celle-ci de manière indépendante du système en cours d'exécution puis la synchroniser par la suite.

Ceci permet par exemple d'introduire des étapes de vérification avant déploiement mais surtout de ne pas contraindre les capacités de modification du M@R avec des contraintes liées à la plate-forme d'exécution. Par exemple si un composant A dépend d'un autre B, la plate-forme d'exécution va exiger que A soit déployé avant B. A l'inverse, l'ajout de B avant A dans le modèle n'est pas soumis aux mêmes contraintes d'ordres si l'application de l'adaptation se fait après les deux ajouts. Les contraintes d'une couche réflexive liée à sa plate-forme d'exécution restreignent alors toutes les approches de génération automatique d'adaptations. A l'inverse en retardant ces contraintes au moment de l'application du modèle réflexif, les approches génératives peuvent ainsi manipuler le modèle dans n'importe quel ordre.

FIGURE 2.2 – Illustration du processus Model@Runtime



Cette capacité à désynchroniser le modèle réflexif et la plate-forme est illustrée par la figure 2.2 et permet d'exploiter à l'exécution (au *runtime*) ce modèle comme une cible pour tous les algorithmes de composition qui servent à calculer à partir de diverses sources (*"feature model"* (dérivation de fonctionnalités), aspects [MBNJ09]) un modèle composé de l'architecture du système. Le système pouvant décider quand synchroniser le modèle, toutes les opérations avant déploiement *(«offline»)* ne sont pas contraintes par le système concret. Plusieurs travaux [KKR+12],[RBD+09] convergent vers l'utilisation de composant logiciel pour encapsuler les notions de cycles de vie des Systèmes Adaptatifs Hétérogènes et Distribués et leurs opérateurs de composition. Utilisant divers paradigmes de conception et de composition pour assembler le système complet, la convergence de ces travaux étaye l'hypothèse à la fois d'une nécessité de diversité de paradigmes de composition mais également de la viabilité d'exploiter ce le bon niveau de granularité pour gérer les cycles de vie des briques applicatives des Systèmes Adaptatifs Hétérogènes et Distribués.

La capacité d'un système adaptatif hétérogène et distribué à établir des modèles qui ne sont pas directement liés à la réalité permet de valider chaque configuration seulement lorsque c'est nécessaire. Si une configuration n'est pas valide, elle est simplement rejetée : il n'y a pas besoin d'effectuer un *roll-back* sur le système, puisqu'il n'a pas été affecté par la construction de ce modèle invalide. Cette validation en ligne fournit un haut-degré de confiance dans le système adaptatif : le processus d'adaptation ne fera jamais migrer le système vers une configuration qui n'a pas été validée. Il est important de noter que ces travaux n'ont pas eu pour but de contribuer sur le processus de validation lui-même. Au lieu de cela, le fait de pouvoir découpler et synchroniser le modèle de réflexion de la réalité permet d'appliquer des techniques de validation existantes et même d'intégrer les futures avancées dans le domaine de la validation.

Cette contribution est présentée dans [MBNJ09]. Ces premiers travaux ont clairement fondé tout le travail autour de l'utilisation des modèles à l'exécution.

2.3 Validation

Le travail de validation a été double. Dans la cadre du cas d'étude présenté pour le déploiement sur la ville de Rennes d'équipements domotiques pour favoriser le maintien à domicile de personnes âgées dépendantes, nous avons modéliser le système, construit un certain nombre d'aspects, modélisé la ligne de produits et montré comment nous gardions le système maîtrisable en suivant l'approche proposée. Tout ce travail a principalement était réalisé uniquement dans une première étape en phase de modélisation. Parallèlement à cela, nous avons construit un environnement d'exécution nommé Entimid[NBF+09] qui nous a permis de déployer ce système dans le laboratoire de domotique de l'université de Rennes 1 et dans trois appartements témoins du projet IDA [3]. Dans le cadre du déploiement au sein du laboratoire, nous avons construit un système ayant le même degré de variabilité que celui proposé dans les motivations. Le déploiement au sein des appartements témoins a été plus simpliste du fait entre autres d'un nombre d'équipements plus limité.

Ce premier travail, directement aligné, avec mes perspectives de thèse a structuré l'ensemble de mon activité de recherche en posant les bases pour la conception et l'administration de systèmes complexes, l'ingénierie multi-points de vue, la modélisation de la variabilité, la composition logicielle et la modélisation à l'exécution.

3. http ://www.loustic.net/ida

Chapitre 3

Améliorations des techniques de modélisation pour leur utilisation à l'exécution dans l'hypothèse d'un monde ouvert [SMM+12, CBJ10, FFBA+14, FNM+12]

Ce chapitre résume quatre contributions pour améliorer l'utilisation des techniques de modélisation sous l'hypothèse du monde ouvert. La première vise à montrer comment diminuer le couplage entre un méta-modèle et un ensemble de transformations, d'outils, ou de cadres de modélisation afin de diminuer la fragilité des ces derniers face à l'évolution du méta-modèle auquel ils sont liés. La deuxième résume un travail mené pour permettre l'utilisation de techniques de composition de modèles dans le cadre de l'intéropérabilité de sytèmes patrimoniaux. Le troisième relate les recherches menées dans le cadre de la modélisation de la variabilité. Enfin la dernière présente les limites du cadre de modélisation de référence à savoir *Eclipse Modelling Framework (EMF)* pour une utilisation à l'exécution des techniques de modélisation et présente ensuite une un cadre de référence permettant l'utilisation de modèles à l'exécution. Les publications supports à ce chapitre sont respectivement [SMM+12, CBJ10, FFBA+14, FNM+12]

Sommaire

3.1 Vers une diminution du couplage par rapport au méta-modèle **29**
3.1.1 Contexte et problématique 29
3.1.2 Contribution 29
3.1.3 Validation 30
3.2 Composition logicielle **31**
3.2.1 Contexte et problématique 31
3.2.2 Contribution 31

3.2.3 Validation 32
3.3 Gestion de la variabilité **33**
3.3.1 Contexte et problématique 33
3.3.2 Contribution 33
3.3.3 Validation 35
3.4 Un cadre de modélisation pour son utilisation à l'exécution **37**
3.4.1 Contexte et problématique 37
3.4.2 Contributions 38
3.4.3 Validation 40

Le chapitre précédent a montré une vision pour la gestion de la construction et de l'administration de systèmes adaptatifs hétérogènes et distribués autour de quatre grands points de vue définis à l'aide de méta-modèles dédiés offrant une abstraction pour capturer :

— sa variabilité, qui décrit les différentes fonctionnalités ou caractéristiques du système, et leurs natures (options, alternatives, etc.)
— son environnement, qui décrit les points importants du contexte que nous voulons surveiller.
— sa logique d'adaptation, qui décrit quand le système doit s'adapter. Cela consiste à définir quelles fonctionnalités/caractéristiques (du modèle de variabilité) choisir en fonction du contexte courant.
— son architecture, qui décrit la configuration du système courant en termes de concepts architecturaux.

À partir de ces quatre points de vue, nous proposons la synthèse d'un modèle de configuration utilisé à l'exécution combinant des techniques issues de la modélisation de la variabilité, des techniques avancées de composition de modèles, et des techniques de modélisation à l'exécution. L'étude en profondeur de cette vision nous a amené dès lors à contribuer à la fois *au niveau méta*[1] pour chacun de ces domaines, c'est-à-dire , sans forcément un lien direct avec un modèle de configuration. Ainsi nous proposons des contributions dans le domaine de la modélisation, entre autres, pour

— diminuer le couplage autour du méta-modèle afin de pouvoir utiliser la modélisation dans un contexte plus agile où l'on souhaite pouvoir facilement faire évoluer un méta-modèle sans perdre l'ensemble des artefacts de modélisation déjà construits (modèles, transformation, opérateur de composition, ...).
— permettre l'utilisation de techniques de compositions de modèles pour l'intégration des systèmes possédant du code patrimonial. Ceci afin de permettre l'utilisation de la modélisation sous l'hypothèse du monde ouvert dans lequel un sous-système ou module peut demander son intégration même si cette intégration n'a pas été prévue à l'origine mais aussi, même si ce système n'a pas été construit à l'aide de techniques de modélisation.
— améliorer l'utilisation de la modélisation de la variabilité dans l'hypothèse d'un monde ouvert, c'est-à-dire , dans l'hypothèse où l'on souhaite modéliser la variabilité pour un ensemble de points de vue qui pourront eux même évoluer.
— permettre l'utilisation d'approches de modélisation à l'exécution en tenant compte des exigences à la fois des systèmes hétérogènes et distribués et des techniques de modélisation.

Les quatre sections suivantes résument ces contributions.

1. De manière générique par rapport au langage de configuration utilisé

3.1 Vers une diminution du couplage des artefacts de modélisation au méta-modèle dans l'ingénierie dirigé par les modèles [SMM+12]

Cette première section présente un travail de synthèse combinant différentes techniques avancées dans le domaine de la modélisation (le typage de modèle et la modélisation par aspects) pour favoriser l'émergence d'artefacts de modélisation réutilisables.

3.1.1 Contexte et problématique

La réutilisation logicielle est une problématique de recherche importante fortement étudiée par la communauté du génie logiciel dans les dernières décennies [BP89, MMM95]. Basili *et al.* [BBM96] ont ainsi montré les avantages liés à la réutilisation logicielle sur la productivité et la qualité dans des systèmes orientés objet. Dans le domaine de l'Ingénierie Dirigée par les Modèles (IDM), qui est généralement construit sur à partir de méta-langages eux mêmes descendant de la programmation par objets, peu de travaux ont été consacrés à la réutilisation [BRR05]. Par conséquent, il en résulte souvent une fragilité importante des approches IDM liée au couplage important entre ces approches et le méta-modèle du domaine sur lequel elles ont été bâties. Par exemple, si beaucoup de langages de programmation ou de modélisation partagent des concepts communs : par exemple, Java, MOF et UML partagent les concepts de classes, de méthodes, d'attributs et d'héritage. Cependant, une transformation de modèle écrite pour UML ne peut pas être réutilisée, par exemple, pour Java parce que leurs arbres de syntaxe abstraite (ou méta-modèles) sont structurellement différents. Si ce problème existe pour écrire une transformation de *refactoring* entre UML et Java, ce problème existe aussi pour toute transformation de modèle (composition, tissage, analyse, ...). Il apparaît aussi à la moindre évolution du méta-modèle, ce qui dans l'hypothèse du monde ouvert doit être une opération simple et potentiellement quotidienne. Ainsi dans [SMM+12], nous posons la question suivante : comment découpler les artefacts manipulant des modèles des méta-modèles associés ? Le but est alors de rendre les artefacts manipulant les méta-modèles plus facilement réutilisables.

3.1.2 Contribution

Pour améliorer la réutilisation au niveau des transformations de modèles, nous proposons une approche combinant la notion d'inférence de types de modèles et la notion de composition de modèles. La vue générale de l'approche est présentée dans la figure 3.1, l'idée générale est d'inférer de manière statique le méta-modèle effectif d'une transformation modèle. Ce méta-modèle effectif nous permet de déduire le type de modèle [SJ07] associé à une transformation de modèles. Dès lors, nous obtenons une transformation de modèle avec un couplage faible par rapport au méta-modèle pour laquelle cette dernière a été initialement conçue. Pour réutiliser cette transformation de modèle, nous proposons ensuite un support à l'alignement de méta-modèle à l'aide d'aspects.

Le besoin d'alignement se justifie généralement par l'absence d'un plus petit méta-modèle commun entre deux méta-modèles conçus sans rechercher à partager une information commune. Ainsi, MOF, UML et Java, s'ils sont tous les trois des méta-modèles de langages objets, n'ont pas réellement de plus petit méta-modèle commun représentant l'essence d'un modèle objet.

L'utilisation de mécanismes d'introduction statique (*intertype definition*) permet ainsi d'expliciter des correspondances structurelles et permet de mettre en place des adaptateurs entre deux méta-modèles non alignés. Par exemple, aligner deux attributs qui diffèrent par leur nom

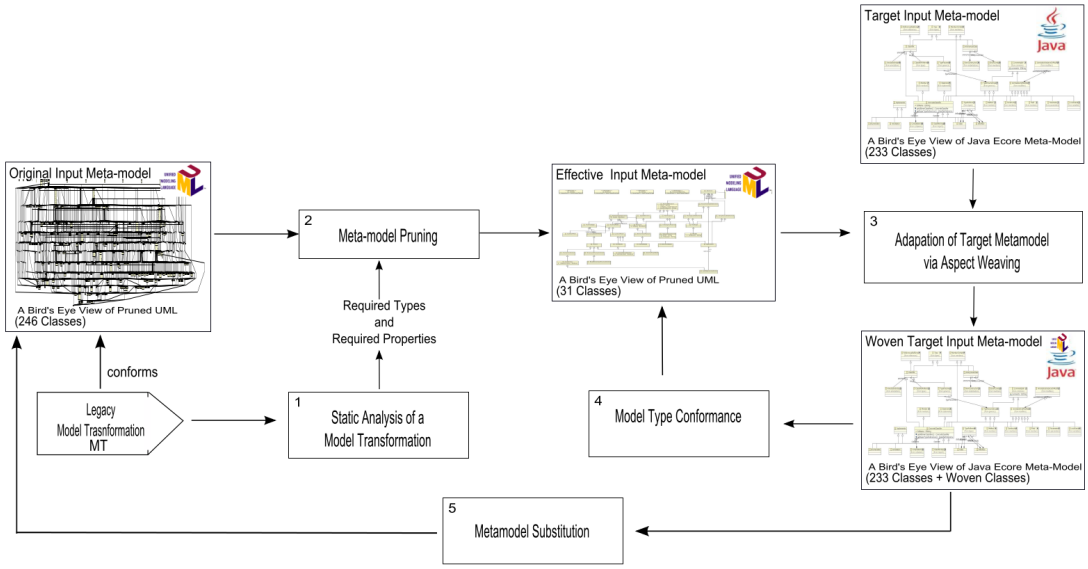


FIGURE 3.1 – Vue générale de l'approche

ou par leur type peut se faire en introduisant un attribut dérivé mettant en œuvre l'adaptation entre ces deux types de données [GHJV94, LQ06].

3.1.3 Validation

Pour illustrer et valider notre approche, nous avons implanté un certain nombre de d'opérations de *refactoring* pour MOF et nous avons montré comment notre approche permettait de diminuer les couplages de ces opérations par rapport au méta-modèle MOF, nous avons ensuite créé à l'aide d'aspects Kermeta [MFJ05] un certain nombre d'adaptateurs pour UML et Java afin de permettre la réutilisation de ces opérations de *refactoring*.

Ce premier travail fournit une contribution pour améliorer les techniques de modélisation sous l'hypothèse du monde ouvert. En effet, en utilisant cette approche une évolution d'un méta-modèle ou l'adaptation d'un artefact de modélisation à un contexte différent se trouvent facilitées, dans un contexte d'ingénierie multi-points de vue comme proposé dans le chapitre 2, cette approche facilite l'évolution des abstractions proposées dans chacun des points de vue et permet de partager certains artefacts de modélisation entre points de vue possédant quelques caractéristiques communes.

3.2 Composition logicielle [CBJ10]

3.2.1 Contexte et problématique

La problématique de composition logicielle est depuis toujours au cœur des activités de conception logicielle. Notion complémentaire à toute approche facilitant la séparation de préoccupations en phase de conception, la composition devient d'autant plus difficile quand nous cherchons à assembler deux modules logiciels n'ayant pas été conçus pour être composables deux à deux. Ainsi dans le monde des systèmes d'information, cette problématique se retrouve généralement quand il s'agit de faire cohabiter deux systèmes qui n'ont pas initialement été conçus pour fonctionner ensemble. Dans le contexte de la problématique d'Intégration d'Application d'Entreprise (EAI), une des préoccupations est généralement l'adaptation de données d'une application vers une autre application. Dès lors la plupart des solutions fournissent des moyens de traduire facilement un format de données (par exemple un format textuel) vers un autre format de données (par exemple un format XML). Le problème n'est généralement pas perçu de la même manière quand il s'agit de composer deux applications au travers de leurs API. Dans ce cadre, l'utilisation d'un patron de conception comme adaptateur [GHJV94] en programmation par objets ou la notion de connecteurs dans le domaine de l'architecture logicielle sont souvent vus comme des moyens d'intégrer des applications d'entreprise par leurs APIs. La difficulté est alors d'éviter la multiplication du nombre de ces adaptateurs liée entre autres aux multiples versions d'API ou de logiciels existantes. Même si l'utilisation d'une API pivot, permet de réduire fortement ce nombre d'adaptateurs, de $2*n*(n-1)$ à $2*(n+1)$ pour n applications à composer, cela laisse malheureusement toujours beaucoup de codes d'adaptation à développer dès que n est grand.

Dans le cadre du projet MOPCOM-I, et au travers de la thèse de Mickaël Clavreul [Cla11], nous avons travaillé sur cette problématique en collaboration avec la société Thomson Multimédia. En effet, dans le domaine de la diffusion de la télévision numérique, le nombre de protocoles d'administration des équipements du réseau met particulièrement en évidence ce problème. Le support des nouveaux produits tout en maintenant la compatibilité avec l'ensemble des équipements existants au travers des différentes versions des standards existants entraîne de nombreuses opérations de maintenance logicielle coûteuses et complexes. La figure 3.2 illustre ce contexte dans le cadre précis de Thomson Multimedia. Pour construire leur outil de management de réseau, Thomson a capitalisé autour d'un modèle pivot nommé XMS, et se retrouve à créer de nombreux adaptateurs pour des protocoles existants du domaine tel que MTEP, SNMP, RS485, ... En outre, ils font face à de nombreuses versions de leur propre protocole XMS mais aussi des protocoles du domaine.

3.2.2 Contribution

Afin de limiter cet effort, nous proposons un processus semi-automatisé afin de limiter les ambiguïtés dans la spécification des adaptateurs, afin de réduire les erreurs dans l'implantation de ces adaptateurs et afin d'automatiser la processus de transformation. Nous explorons comment les modèles, les aspects et l'utilisation de techniques génératives peuvent être utilisées conjointement pour simplifier la création de multiples adaptateurs.

Le processus est fondé sur trois étapes principales : (1) la rétro-ingénierie automatique de concepts pertinents dans des API au sein d'un modèle abstrait ; (2) la définition manuelle des correspondances entre des concepts présents dans des modèles d'API de code patrimoniaux différents à l'aide d'un langage dédié ; (3) la génération automatique d'adaptateurs à l'aide de techniques de programmations par aspects afin de se composer avec les APIs existantes.

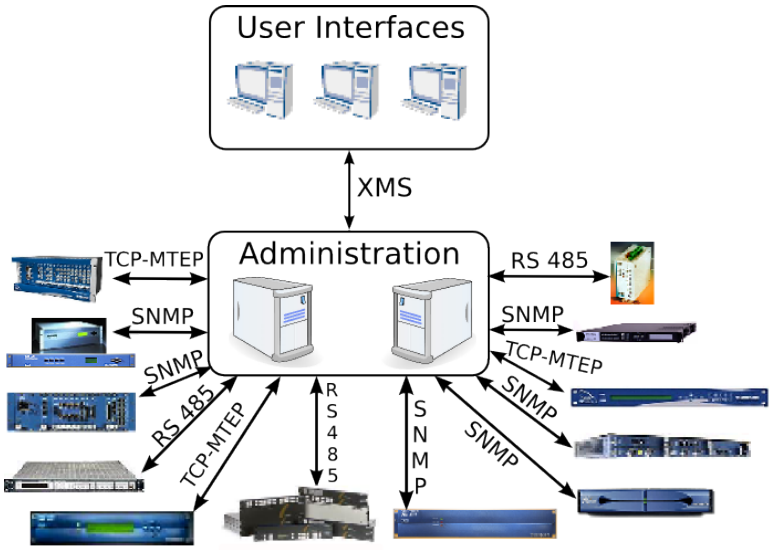


FIGURE 3.2 – Exemple de cas d'applications

Le langage dédié d'expression de correspondance s'appelle ModMap [2]. Il permet d'exprimer de manière graphique les correspondances entre deux modèles d'API modélisés à l'aide d'un diagramme de classes UML.

3.2.3 Validation

Pour valider cette approche, nous avons travaillé en collaboration avec Thomson Multimedia sur le cas d'étude présenté dans les motivations de ce travail. Nous avons montré que le langage dédié était appréhendable par des experts métiers de chez Thomson, nous avons montré que sous certaines conditions le gain en terme de productivité pouvait atteindre 85 % sans coût à l'exécution lié à l'usage de la programmation par aspect pour tisser les adaptateurs. Cet absence de coût à l'exécution s'explique par le fait que nous utilisons uniquement des mécanismes d'introduction statique et aucun mécanisme d'indirection du flot d'exécution. Suite à cette validation, nous avons effectué une revue de la littérature scientifique sur la composition de modèles et proposé une classification des approches de composition existantes ainsi qu'une formalisation abstraite de la notion de composition de modèles. Ces derniers points ont été décrits dans la thèse de Mickaël Clavreul [Cla11].

Ce travail permet de faciliter l'intégration de code ou de système patrimonial dans des démarches de modélisation. Dans le cadre de notre démarche où nous proposons de modéliser un système adaptatif à partir de ***quatre points de vue*** *(voir chapitre 2), ce type d'approche est particulièrement utile pour intégrer, par exemple, des frameworks de monitoring de systèmes rarement construits à partir de techniques de modélisation.*

2. http://www.kermeta.org/mdk/ModMap/

3.3 Gestion de la variabilité [FFBA+14]

3.3.1 Contexte et problématique

Dans le cadre de notre démarche où nous proposons de modéliser un système adaptatifs à partir de **quatre points de vue** (voir chapitre 2). Un des points de vue concerne la modélisation de la variabilité du système. De nombreuses approches cherchent à combiner des techniques de modélisation et des techniques de gestion de la variabilité. On regroupe généralement ces approches sous la bannière *Model-based Software Product Line (MSPL)*. L'objectif est de capitaliser la variabilité dans les artefacts de modélisations et de dériver automatiquement des artefacts de modélisation spécialisés correspondant à un produit d'une famille de produits. Ce type d'approche utilise généralement une des deux techniques pour modéliser la variabilité : soit une extension du méta-modèle pour capturer les informations de la variabilité [GVM+12]. On parle alors d'approches amalgamées, soit une approche qui utilise un modèle pour capturer les informations de variabilité exprimées alors dans un formalisme issu des diagrammes de *features* dans lequel ces *features* sont liées à des artefacts de modélisations conforme à un méta-modèle de domaine. L'expression de ces liaisons ou correspondances entre le modèle de *feature* et le modèle du domaine pouvant parfois être complexe, il peut-être exprimé à l'aide d'un formalisme dédié nommé sous le terme de modèle de réalisation.

L'utilisation de la première solution entraînant une modification des différents langages de domaine amène un certain nombre de lacunes pour un transfert vers le domaine industriel, entres autres car elle pose la question de la compatibilité de l'ensemble des outils existant autour d'un langage d'un domaine. De fait, la deuxième solution semble être celle qui s'impose. L'utilisation d'un langage générique pour capturer la variabilité de manière orthogonale à la modélisation d'un domaine métier particulier entraîne un certain nombre de challenges scientifiques [HHSD10] parmi lesquels :

— Est il nécessaire de spécialiser un langage d'expression de la variabilité pour un domaine dédié ? Si oui, comment spécialiser de manière élégante cette sémantique ?

— Le domaine de modélisation, si il est déjà vaste pour un concepteur d'applications sans prise en compte de la variabilité, devient difficilement maîtrisable par un concepteur ou un architecte devant construire une ligne de produits. En effet, le nombre de produits d'une ligne de produits croit rapidement, chacun de ces produits est généralement soumis à un ensemble de contraintes du domaine métier. Comment détecter des produits erronés par construction sans devoir dériver l'ensemble des produits possibles ?

3.3.2 Contribution

Dans le cadre de la thèse de Bosco Ferreria Filho, nous avons travaillé en étroite collaboration avec la société Thales sur ces problématiques autour d'une initiative de standard nommé CVL pour *Common Variability Language*. CVL présenté dans la figure 3.3, propose une modélisation orthogonale de la variabilité. Ce langage est construit autour de trois types de modèles permettant de capturer : i) les points de variation possibles (modèle de choix), la définition des actions à réaliser sur les éléments de modèle du domaine métier lors de la dérivation associé à chacun de ces choix (modèle de réalisation), iii) les décisions associées à la construction d'un produit (modèle de décision).

Nous avons démontré que si nous considérons un modèle de variabilité CVL comme un triplet $(V \times R \times D)$ représentant respectivement le modèle de variabilité V, le modèle de réalisation R et le modèle de domaine D pour lequel nous modélisons la variabilité, il était fréquent d'obtenir des cas où V est valide, R est valide et D est valide mais une configuration C (un modèle de décision) particulière entraîne suite à une dérivation un produit incorrect. Le modèle CVL est

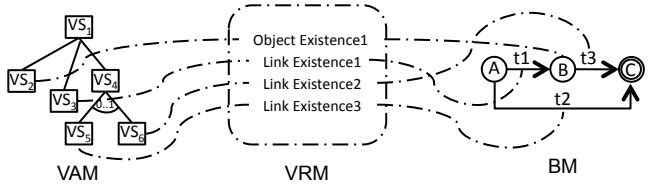


FIGURE 3.3 – Exemple de modèle CVL appliqué à un modèle de domaine de machine à états finie

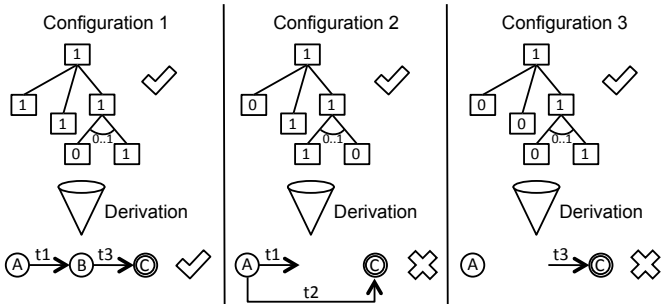


FIGURE 3.4 – Illustration schématique du problème

dit alors **incohérent**. À partir de ce constat, l'effort demandé au concepteur d'une famille de produit devient irréaliste. En effet, tout atelier de modélisation à l'état de l'art indiquera par défaut que la modélisation est correcte, or cette dernière autorisera la construction de produits incorrects prouvant l'incohérence de la modélisation de la ligne de produits. L'apparition de ces incohérences est fortement dépendante des contraintes du domaine métier. Ce constat justifie le besoin d'un cadre de modélisation permettant de détecter ces incohérences au plus tôt. Ce problème est illustré dans les figures 3.3 et 3.4 dans lesquels on schématise un modèle CVL et trois résultats de dérivation possibles en fonction de trois modèles de décision $C1$, $C2$, $C3$.

Dès lors, il est possible de, soit fournir un ensemble de règles de bonnes formations supplémentaires pour interdire des constructions de variabilité pour certains concepts métier afin de détecter ces incohérences, soit spécialiser la sémantique de dérivation en fonction du domaine métier pour garantir par construction la cohérence des produits obtenus.

Avant la mise en place de tels *frameworks*, il est nécessaire d'assister un expert de domaine afin de lui permettre de comprendre quel triplet (en termes de constructions de modèle de variabilité, constructions de modèle de réalisation, constructions de modèle de domaine) peut entraîner des incohérences.

Dans ce cadre, nous proposons un processus automatisé qui explore de manière aléatoire l'espace de modélisation. Le but ést de trouver, à partir d'un domaine métier défini par son méta-modèle et ses contraintes métiers, des exemples de modèles CVLs dont le modèle de variabilité, le modèle de réalisation et le modèle de domaine sont valides mais pour lesquels il existe au moins une configuration C qui crée un produit (c'est-à-dire un modèle métier) invalide. Nous appelons de tels modèles CVL des contre-exemples. La vue générale de ce processus est illustrée au travers des figures 3.5 et 3.6.

Ces contre-exemples permettent de guider les experts de domaine pour spécialiser la sémantique de dérivation ou spécifier des règles métier permettant d'interdire la création de modèles CVL inconsistants.

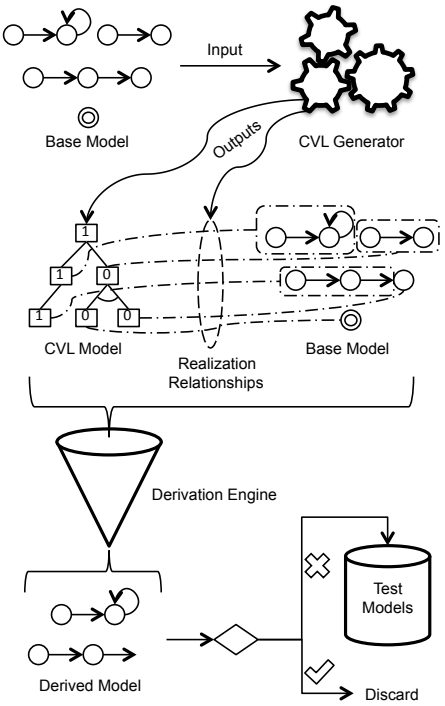


FIGURE 3.5 – Vue générale du processus automatisé pour la création de contre-exemples

3.3.3 Validation

Afin de valider cette problématique, nous avons exécuté notre processus sur trois domaines métier, respectivement un méta-modèle de machine à états finie (FSM pour *Finite State Machine*) composé de 3 méta-classes, 1 *datatype* et 4 règles métiers exprimées à l'aide du langage OCL. un méta-modèle représentant une implémentation d'*essential MOF* [OMG06] à savoir Ecore [SBMP08] composé de 20 méta-classes, 33 *datatypes* et 91 règles de validation et enfin UML2.x [RJB04] contenant 247 méta-classes, 17 *datatypes* et 684 règles de validation.

Cette validation vise à répondre à quatre questions de recherche.

1. Tout d'abord, est il facile de générer de tels contre-exemples pour différents domaines ? Pour répondre à cette question, nous avons cherché à savoir le temps moyen pour générer 100 contre exemples pour ces différents domaines sur une machine standard (processeur Intel Core I7 2ème génération - 16Go de mémoire tournant sous linux 64bit, le générateur de contre exemple est écrit à l'aide du langage Scala en version 2.9.3 et est exécuté à l'aide d'une JRE 7 d'Oracle.
2. Quelle est la corrélation entre la complexité pour trouver un contre exemple et la complexité du domaine métier sur lequel on veut exprimer de la variabilité ?
3. Quels sont les types d'erreurs les plus fréquentes ? Des violations au niveau de la relation de conformance entre un modèle métier et son méta-modèle ou des violations des contraintes du domaine métier.
4. Enfin, est il possible d'écrire des règles permettant d'interdire certaines formes de contre-exemple ou est il possible de spécialiser la sémantique de dérivation d'un domaine pour éviter la création de produits finaux invalides ?

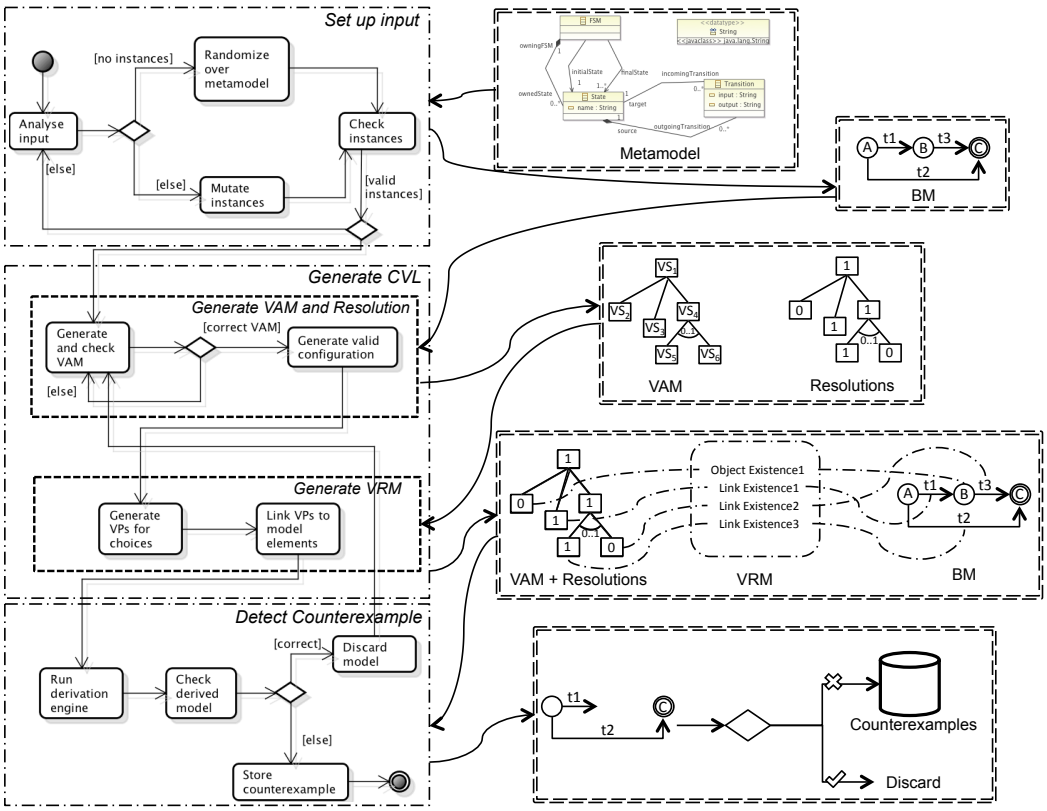


FIGURE 3.6 – Vue détaillée du processus automatisé pour la création de contre-exemples

Pour la première question, nous avons démontré qu'il était possible de générer dans un temps raisonnable ces 100 contre-exemples (respectivement 5742s pour le méta-modèle des machines à états et 12625s pour *ecore*.) Comme cette exploration est facilement parallélisable, nous pouvons imaginer diminuer fortement ces temps.

Dans le cadre de la deuxième question, nous avons montré que si dans le cas de FSM, nous générons 1 contre exemple pour 5 modèles CVL valides, ce ratio tombe à 1 pour 4 pour ecore et 2 pour 1 pour UML. Ainsi, par extrapolation, nous pouvons renforcer l'hypothèse selon laquelle plus le domaine métier est complexe, plus le risque de produire un modèle CVL inconsistant est important.

Dans le cadre de la troisième question, nous avons observé les 100 contre-exemples générés pour chaque domaine. Dans le cadre de l'expérimentation sur FSM, nous constatons que 90 modèles CVL générés entraînent la création d'un produit qui viole une contrainte du domaine et 10 entraînent la création d'un modèle non conforme à son méta-modèle. Ces chiffres passent respectivement à 36 et 64 pour ecore et 78 pour 22 pour UML. Cette expérience ne fait pas ressortir clairement de corrélation entre la complexité du domaine métier ou le nombre de contraintes métier et le type d'erreur introduite par une modélisation de la variabilité. Ceci peut certainement s'expliquer par l'existence de différents styles de modélisation [CBS12], certaines contraintes peuvent par exemple être réifiées dans le méta-modèle ou écrites sous forme d'expressions OCL.

Enfin pour la dernière question, nous avons cherché à montrer qu'il était possible après une analyse des contres-exemples générés de définir une spécialisation de la sémantique de dérivation

ou la définition de règles de validation propre à CVL et un domaine métier permettant de réduire fortement le risque d'apparition de contre-exemple. Ainsi dans le cadre du domaine des FSMs, nous avons pu suite à l'écriture de quatre règles de validation passer d'un *ratio* de génération de 1 contre-exemple pour 5 modèles CVL valides à 1 pour 20.

D'autres validations ont fait suite à ce travail, dans le cadre d'un partenariat avec Thales, nous avons appliqué cette méthodologie sur leur langage d'ingénierie système. Dans ce cadre, nous avons analysé finement le type de contre-exemple obtenu pour spécialiser la sémantique de dérivation de CVL dans le cadre de ce langage d'ingénierie système. Afin de faire face à la taille du domaine de modélisation de langage d'ingénierie système, nous avons adapté le processus de génération pour n'utiliser que des constructions du langage de domaine habituellement utilisé. Pour ce faire, nous sommes partis d'un ensemble d'exemples métier et nous avons introduit des opérateurs de mutation simples pour augmenter ces modèles de domaine servant de base au processus de dérivation.

Dans la continuité de ce travail, nous avons dès lors proposé un cadre de modélisation dédié à la spécialisation de la sémantique opérationnelle de CVL [FBLNJ12].

Ce travail nous a permis de faciliter l'utilisation d'approches de modélisation de la variabilité dans un contexte d'ingénierie multi-vues comme celui proposé dans le chapitre 2. Ainsi, l'approche proposée permet de simplifier le travail pour un expert de domaine cherchant à modéliser un point de vue supplémentaire ou cherchant à faire évoluer un point de vue existants en permettant la spécialisation de la modélisation de la variabilité pour ce nouveau point de vue.

3.4 Un cadre de modélisation pour son utilisation à l'exécution [FNM+12]

Le quatrième axe sur lequel nous avons contribué de manière générique se concentre autour d'un cadre de modélisation pour un usage des modèles à l'exécution. Cette section détaille le contexte mais surtout la problématique associée à cette recherche et synthétise les principaux résultat obtenus.

3.4.1 Contexte et problématique

L'utilisation de techniques de modélisation à l'exécution entraîne naturellement l'usage de technologies issue de l'IDM dans des systèmes. Le bénéfice obtenu est alors de garantir une interopérabilité entre les outils de conception et les outils d'administration d'un système à l'exécution. Parmi, les cadres de développement qui se sont imposés dans la communauté se trouve EMF (pour *Eclipse modelling framework*). Ainsi, par exemple, Frascati [SMR+11] (mise en œuvre de du standard SCA Service Component Architecture) utilise EMF pour la mise en œuvre du langage de configuration associé à ce standard. Ceci pose plusieurs problèmes, les cadres de développement comme EMF ont été généralement pensés uniquement pour une utilisation dans le cadre d'activité de conception, construits au cœur d'environnements de développement intégrés (IDE). En outre, certaines opérations importantes à l'exécution comme la capacité à conserver un historique des évolutions, fournir différents formats de sérialisation, fournir des API pour différents langages pour manipuler ces modèles, *etc.* n'ont pas été prévues dans la conception de ces cadres de développement.

La maturation des techniques de modélisation à l'exécution passe nécessairement par la prise en compte des exigences propres aux techniques de modélisation à l'exécution. Dans ces travaux, nous avons cherché à éliciter les exigences liées à l'utilisation des modèles à l'exécution. À partir de cette étude nous avons travaillé à construire un cadre de modélisation nommé KMF pour *Kevoree Modelling Framework* dont le but est de conserver une compatibilité avec EMF tout en répondant aux exigences liées à l'utilisation des modèles à l'exécution.

3.4.2 Contributions

Dans le cadre de ce travail, nous proposons trois sous-contributions : une élicitation des exigences pour un cadre de modélisation à l'exécution, une critique constructive du cadre de modélisation de référence, à savoir EMF, face à ces exigences, et un cadre de référence conçu pour faire face à ces exigences.

3.4.2.1 Élicitation des exigences pour un cadre de modélisation utilisable à l'exécution

L'analyse du domaine de la modélisation à l'exécution, discutée dans le cadre de [FNM+12], nous permet de lister les exigences suivantes :

Faible empreinte mémoire. L'empreinte de mémoire d'un cadre de modélisation à l'exécution détermine les types de nœuds capables d'embarquer ce type de technologie. Au plus une couche de modélisation à l'exécution est consommatrice de mémoire, au plus il sera difficile de la déployer sur des équipements contraints en termes de ressources. Si des techniques de chargement paresseux (*lazy loading*) peuvent être utilisées pour réduire cette consommation mémoire, elle limite alors le type d'équipement pouvant raisonner en profondeur sur le modèle. Dans ce cas, seulement quelques équipements puissants peuvent raisonner et prendre des décisions pour tous les équipements plus contraints. Ceci empêcherait une décentralisation forte du processus d'adaptation.

Limitation des dépendances. Un cadre de développement à l'exécution doit être le plus possible auto-contenu. Un nombre de dépendances important augmente le risque d'embarquer à l'exécution des fonctionnalités non essentielles ce qui entraîne là aussi un surcoût en termes de mémoire consommée. En outre, de nombreuses dépendances vont entraîner un surcoût en temps pour l'initialisation ou la mise à jour d'un nœud dans un environnement distribué. Il sera en effet nécessaire de télécharger et d'installer l'ensemble de ces dépendances ou des mises à jour de ces dépendances.

Thread Safety. Un modèle à l'exécution va généralement s'exécuter dans un cadre où une manipulation simultanée par plusieurs processus légers (exécution multiprocessus, ou multi-threads) est courante. Plusieurs moteurs de raisonnement peuvent travailler sur ce modèle, plusieurs préoccupations, adaptation, surveillance, ...vont se servir du modèle comme base de connaissance. Un cadre de modélisation à l'exécution doit assurer que les multiples threads d'une application puissent accéder et modifier les modèles sans s'inquiéter des détails d'accès concurrents. Particulièrement il doit être possible de naviguer en parallèle sur les ensembles de modèle-éléments du modèle pour mettre en œuvre de manière efficace et sur des opérations de validation ou de simulation.

Un ensemble d'opérateurs de manipulation de modèles spécialisé pour l'utilisation des modèles à l'exécution. Le cadre classique de modélisation à l'exécution entraîne le besoin de charger et sauvegarder un modèle mais aussi très régulièrement de le cloner, d'appliquer des opérations de calcul de différence et de fusion, ou de conserver un historique de son évolution. L'ensemble de ces opérations constituant l'algèbre de base nécessaire à une utilisation des modèles à l'exécution, un cadre de modélisation à l'exécution doit fournir par construction un support efficace pour ses algorithmes. En outre certains de ces algorithmes ayant besoin d'une spécialisation pour un domaine dédié, le cadre de modélisation à l'exécution doit être facilement extensible pour spécialiser certains de ces opérateurs.

Ouverture à de multiples langages de programmations Les modèles servant d'abstraction à l'exécution. Il est nécessaire d'ouvrir le framework de modélisation à de nombreux langages de programmation. L'utilisation des modèles à l'exécution n'a pas de raison de se cantonner à un univers Java, Python ou Ruby. Il est nécessaire de pouvoir manipuler ces abstractions avec des APIs présentes dans différents langages de programmations en suivant les habitudes et les standards de développements de ces communautés.

Conserver une compatibilité avec l'éco-système de modélisation Il est enfin important de conserver une compatibilité transparente avec l'éco-système de modélisation actuel entres autres autour d'EMF afin de conserver une compatibilité avec les outils de conception existants.Ainsi, un simulateur graphique utilisé pour la conception de machines à états finies doit pouvoir servir à surveiller ou déboguer un système conservant ces machines à états à l'exécution [GVdHT09, BHB09].

3.4.2.2 EMF : avantages et inconvénients

L'utilisation directe d'un cadre de modélisation comme EMF à l'exécution offre plusieurs avantages. En effet, nous obtenons de base une compatibilité parfaite avec l'espace de modélisation pour lequel EMF a été pensé. EMF fournit des mécanismes de chargement paresseux permettant de réduire le coût en mémoire des modèles. EMF fournit enfin un support de certains opérateurs comme le chargement ou la sauvegarde ou le clone de modèles.

Néanmoins, plusieurs points sont particulièrement problématiques pour une utilisation à l'exécution. Tout d'abord, EMF a fondamentalement été conçu pour fonctionner au sein d'eclipse. Cela se traduit au regard de ses nombreuses dépendances (voir Figure 3.8). Ces dépendances, qui ne coûtent pas grand chose dans le cadre d'Eclipse car elles font souvent partie du cœur de la plate-forme, n'ont aucun sens à l'exécution. Ainsi sur l'exemple d'un méta-modèle de machine à états finie de quatre méta-classes (voir figure 3.7), nous constatons que le volume de dépendances à embarques à l'exécution dépasse 15MB pour 55kB de code généré. Certains choix de conception comme l'utilisation de registres statiques pour accéder aux classes utilitaires ou la non protection des accès concurrents sont particulièrement problématiques dans le contexte de l'usage des modèles à l'exécution. Enfin les opérateurs de base fournis comme le chargement, la sauvegarde ou le clone d'un modèle sont coûteux en temps et en mémoire.

3.4.2.3 KMF : un cadre de modélisation pour les modèles à l'exécution

Pour contourner ces limitations, nous avons choisi de proposer notre propre cadre de modélisation à l'exécution. Dans ce cadre, nous avons principalement travaillé à conserver une compatibilité avec EMF et nous fournissons une implémentation auto-contenue dans différents langages comme Javascript, Java, C++, bientôt Go et Python. Un effort important de conception combinant des techniques de *copy on write*, lié à l'utilisation du patron de conception

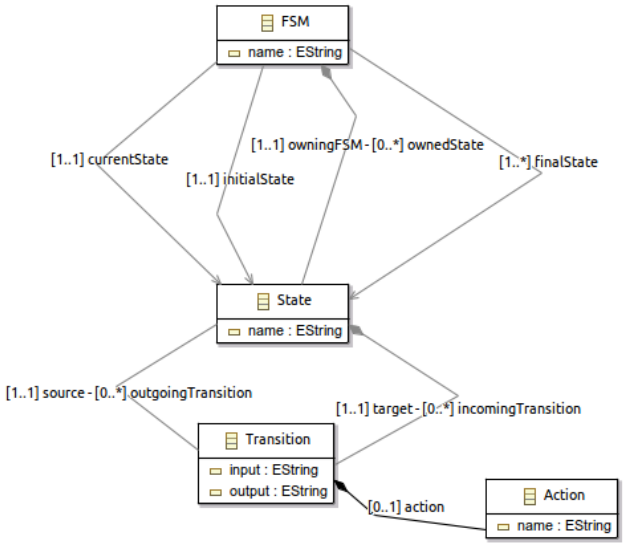


FIGURE 3.7 – Méta-modèle de machine à états finie utilisé dans le cadre de ces expériences

poids-mouche [GHJV94] ont permis de proposer de base un support efficace pour les opérations de chargement, de sauvegarde, de clone, de calcul de différences ou de fusions de modèles. Récemment [HFN+14], nous avons ajouté la notion de version pour permettre de tracer l'évolution d'un modèle à l'exécution.

3.4.3 Validation

La validation s'est principalement faite au regard d'une comparaison en terme de performance et d'empreinte mémoire par rapport à EMF. Ainsi, dans [FNM+12], nous avons montré que KMF allait 1.2 fois plus vite pour la création de modèle, 9 fois plus vite pour le clone de modèle et 2.66 fois plus vite pour la sauvegarde de modèle. Il permettait en outre de consommer 1.7 fois moins de mémoire pour le même modèle en mémoire. Différentes optimisations maintenant présentes dans KMF ont permis d'améliorer encore ces gains [FNM+14].

Ce travail nous a permis de montrer l'existence d'exigences spécifiques à un cadre de modélisation utilisable dans le contexte de systèmes adaptables distribués et hétérogènes. La critique d'un des cadres de modélisation existants et la proposition de notre propre cadre modélisation pour une utilisation à l'exécution tend à rendre réaliste l'approche proposée dans le chapitre 2.

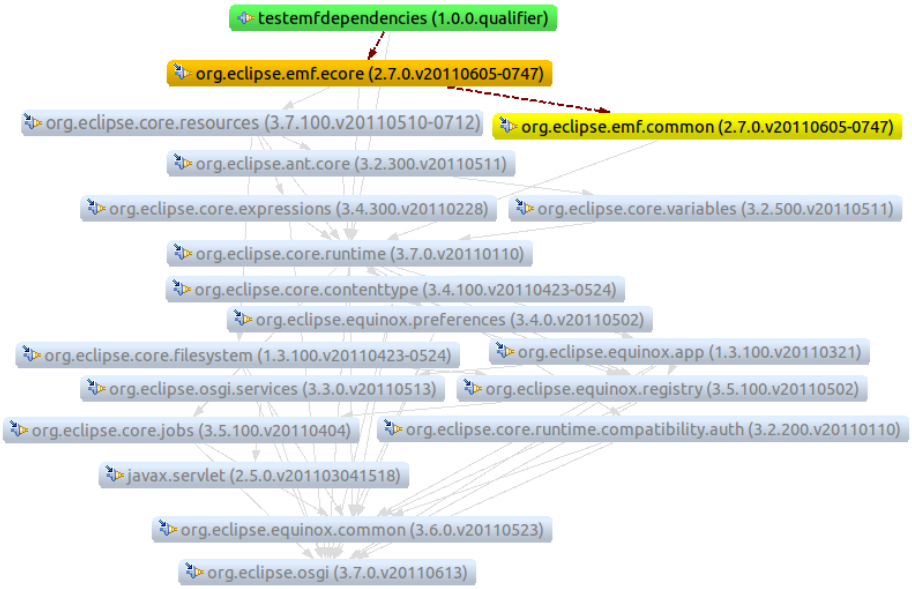


FIGURE 3.8 – Dependencies for each new metamodel generated code

Chapitre 4

Modèle à l'exécution pour la gestion de systèmes adaptatifs hétérogènes et distribués[FDP+12b, FMF+12]

Sommaire

4.1 Contexte et problématique **43**
4.2 Contributions **44**
4.2.1 Les caractéristiques de Kevoree 44
4.2.2 Quelles abstractions ? 46
4.2.3 Support de l'hétérogénéité 51
4.2.4 Extensibilité de Kevoree 51

Ce chapitre présente le travail mené pour comprendre comment la vision autour du models@runtime définie dans la thèse de Brice Morin pouvait s'appliquer au domaine des « Systèmes Adaptatifs Hétérogènes et Distribués ». Cela nous a amené à travailler sur deux points importants résumés dans RQ1 et RQ4 : quelles abstractions pour modéliser la problématique de configuration d'un tels systèmes ? Comment permettre le partage et la dissémination d'un modèle de configuration globale à l'état du système dans un système distribué caractérisé par l'absence d'état global ? Ce chapitre résume la conception de différents éléments de Kevoree en s'intéressant en particulier à la définition de la synchronisation du modèle dans les systèmes distribués. Ce travail a été principalement mené dans le cadre de la thèse de François Fouquet [Fou13]. Ce chapitre ne résume pas de validation, plusieurs validations liées à ce travail sont présentées dans la deuxième partie de ce manuscrit.

4.1 Contexte et problématique

La complexité croissante des systèmes d'information modernes a motivé l'apparition de nouveaux paradigmes (objets, composants, services, etc), permettant de faciliter la réutilisation et maîtriser la complexité liée à la construction de tels systèmes en permettant une meilleure

séparation des préoccupations et en offrant des mécanismes de composition avancés. Tout système moderne, construit de manière modulaire est, aussi, généralement reconfigurable afin de minimiser les temps d'arrêt dûs aux évolutions ou à la maintenance du système. Afin de garantir des propriétés non fonctionnelles (par exemple, maintien du temps de réponse malgré un nombre croissant de requêtes), ces systèmes sont également amenés à être distribués sur différentes ressources de calcul (grilles). Outre l'apport en puissance de calcul, la distribution peut également intervenir pour distribuer une tâche sur des nœuds aux propriétés spécifiques. C'est le cas pour les terminaux mobiles proches des utilisateurs ou encore des objets et capteurs connectés proches physiquement du contexte de mesure. La conception des systèmes modernes de manière modulaire à partir de briques logicielles génériques et réutilisables entraîne une réelles complexité pour la configuration de ces systèmes. En outre, si l'on souhaite rendre un système adaptable afin de permettre son évolution continue, cette logique de configuration doit être connue et partagée. Dès lors, deux challenges importants se posent autour des questions de recherche RQ1 et RQ4. Quelles abstractions sont pertinentes pour ce modèle de configuration, comment l'articuler avec les autres points de vue du systèmes ? Comment un modèle de configuration qui représente l'état du système peut être propagé à l'ensemble des nœuds de calcul ? Le maintien de la cohérence et le partage de cet état sont rendus particulièrement difficiles en cas de connexions sporadiques inhérentes à la distribution, pouvant amener des sous-systèmes à diverger. Comment modéliser le mode de dissémination de cette couche de réflexion afin que ce mode de dissémination devienne une des fonctionnalités reconfigurables du système ?

4.2 Contributions

Cette section revient en détail sur la création d'un cadre de développement d'applications à base de composants pour la conception de systèmes adaptatifs, distribués et hétérogènes nommé Kevoree [1]. Fondé sur le paradigme de modèle à l'exécution (M@R), ce cadre propose un modèle abstrait de configuration pour de tels systèmes permettant la manipulation des différents concepts caractérisant un système distribué. Cette section résume sur un certain nombre de caractéristiques de Kevoree et présente les éléments clé du langage de configuration en insistant en particulier sur la notion de groupe permettant de réifier la sémantique de synchronisation du modèle de configuration entre différents nœuds d'exécution. Enfin, nous présentons en détails les différents points d'extension proposés pour permettre de spécialiser Kevoree à ces besoins. La validation liée à la question de recherche RQ4 est détaillée dans le chapitre 5. La validation liée à la question de recherche RQ1 se retrouve discuter généralement dans la partie 2.

4.2.1 Les caractéristiques de Kevoree

Kevoree a pour objectif de fournir une abstraction pour les systèmes distribués afin de pouvoir manipuler les principaux concepts de ce genre de système et vise à faciliter la gestion de l'adaptation pour ces systèmes. Pour ce faire, Kevoree propose de supporter la synchronisation et désynchronisation entre le modèle réflexif et le système en cours d'exécution, la capacité à représenter la distribution du système, la séparation entre les composants métier et leurs interactions, la dissémination des reconfigurations ainsi que l'hétérogénéité des ressources sur lesquelles s'exécute le système.

1. http://kevoree.org

4.2.1.1 Séparation entre composants métier et interaction (communication)

Une application distribuée est composée de code métier spécifique mais aussi de code de communication. Contrairement au code métier, le code de communication ne possède pas obligatoirement de spécificités dues à l'application. De ce fait, il est intéressant de pouvoir décorréler le code métier de celui chargé de la communication permettant ainsi la réutilisation des différentes briques logicielles. Cela permet de simplifier la conception de composant métier en masquant la problématique de communication. De plus, l'adaptation des moyens de communication entre composants selon le contexte est une exigence d'une application distribuée.

4.2.1.2 Gestion de la distribution

Pour représenter un système distribué, les caractéristiques de distribution se doivent d'être modélisées et manipulables [Gue99] afin par exemple de permettre des adaptations de la distribution des différents éléments d'une application sur l'ensemble des systèmes d'exécution.

4.2.1.3 Désynchronisation entre le modèle réflexif et le système correspondant

Kevoree étant basé sur les techniques de modèle à l'exécution, la définition d'une adaptation s'effectue par la définition d'un modèle qui sera soumis au processus de modèle à l'exécution (voir section 2.2) afin d'être validé avant d'être appliqué. C'est ce processus qui permet d'avoir un modèle réflexif désynchronisé du système en cours d'exécution.

Le fait de pouvoir désynchroniser le modèle réflexif du système en cours d'exécution permet de valider une configuration avant sa mise en place. Cette validation permet notamment de s'assurer de la cohérence de la configuration afin d'éviter un état instable du système voire une panne de celui-ci. Cette caractéristique est d'autant plus importante dans le cadre de système distribué afin d'éviter l'adaptation de certains nœuds du système alors que d'autres ne peuvent pas mettre en place cette nouvelle configuration.

4.2.1.4 Dissémination des adaptations

Une fois qu'une adaptation est soumise (voir section 2.2), chacun des nœuds faisant partie du système se doit d'être notifié afin qu'elle soit prise en compte sur l'ensemble du système. Cependant, un système distribué ne permet pas forcément d'assurer une communication permanente entre les différents nœuds notamment dans le cadre de réseaux sporadiques, c'est-à-dire de réseaux sujet à de fréquentes erreurs et/ou de fréquentes déconnections. La prise en compte de ces contraintes réseaux dans la dissémination des adaptations est donc nécessaire pour assurer que le système évolue de manière cohérente. Ainsi selon les différents types de communication existants entre les nœuds, la synchronisation peut se faire de différentes manières.

4.2.1.5 Hétérogénéité des systèmes d'exécution

Les systèmes distribués peuvent être composés de nombreux types de plates-formes d'exécution que ce soit des plates-formes mobiles comme des *smartphones*, des PCs, des serveurs ou d'appareils embarqués tels que des *Arduinos*[2]. Il est donc nécessaire que le modèle Kevoree permette de différencier ces différentes plates-formes d'exécution qui ont des caractéristiques spécifiques.

2. http://arduino.cc

4.2.2 Quelles abstractions ?

Nous décrivons ici les différents paradigmes de l'approche Kevoree qui permettent l'adaptation dynamique de logiciels. Nous allons tout d'abord, en section 4.2.2.1, présenter les paradigmes utilisés pour représenter un système. Puis, en section 4.2.2.2, nous allons présenter les paradigmes intégrés à Kevoree. Afin d'expliquer à quoi correspondent ces différents concepts, nous suivrons un exemple jouet d'un système de messagerie instantanée utilisée par deux utilisateurs distants (voir figure 4.1).

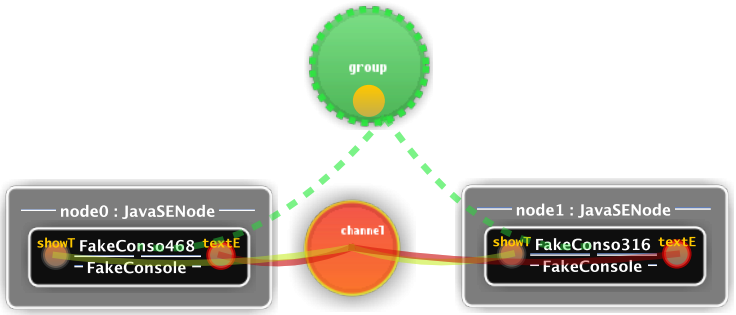


FIGURE 4.1 – Exemple d'application Kevoree

4.2.2.1 Paradigmes de modélisation

L'abstraction proposée par le modèle de configuration de Kevoree est construit à partir de cinq concepts clé. Les trois premiers : composant, canaux de communication, et nœud sont classiques et se retrouvent dans la plupart des langages de description d'architecture [HRPL+95, MT00]. La notion de groupe, notion de premier ordre dans notre langage, permet de réifier la sémantique de dissémination du modèle de configuration entre nœuds. Enfin, différents artefacts permettent de modéliser la topologie de communication réseau et ses caractéristiques.

Composants Les composants Kevoree définissent un contrat d'interface [BJPW99]. Ce contrat définit un ensemble de fonctionnalités fournies, requises et identifiées de manière unique par un port. Un composant définit donc une ou plusieurs fonctionnalités offertes au système. Un composant A est défini comme substituable à un autre composant B si A possède au moins l'ensemble des ports que possède le composant B. Cette caractéristique offre la possibilité d'adapter le système en remplaçant un composant par un autre. Cette adaptation peut se faire en fonction de propriétés non fonctionnelles et permet de reconfigurer les applications durant la phase de conception et durant l'exécution de celles-ci tout en maintenant les fonctionnalités requises.

Dans l'exemple de l'application de messagerie instantanée, les boites noires correspondent aux composants avec un port d'entrée et un port de sortie.

Canaux de communication (*Channels*) En plus des composants, une application définit aussi les relations entre ses composants afin de les faire collaborer. Ces relations sont représentées sous forme de canaux (channels) qui encapsulent les sémantiques de communication entre composants.

Dans l'exemple, nous avons un canal de communication appelé "*channel*" qui permet de connecter l'ensemble des ports des deux composants.

Nœuds Un système distribué est caractérisé par un ensemble de nœuds de calcul. Chaque nœud peut donc héberger un ensemble de processus métier (*composants*) eux-mêmes interconnectés entre eux par des canaux de communication (*channels*) formant ainsi une application. Un nœud est donc un conteneur qui fournit un niveau d'isolation et qui est responsable localement de la synchronisation entre son modèle d'architecture et le système qui s'exécute. Cette synchronisation se caractérise par l'exécution des reconfigurations au travers des primitives d'adaptation. L'exécution de l'adaptation est le plus souvent dépendante du support d'exécution. Par exemple, l'instanciation d'un composant sur un nœud Java n'est pas la même que sur un nœud de microcontrôleur en C. C'est pourquoi, si la logique des opérations sur le modèle sont partagés entre nœud, les implémentations des primitives d'adaptation sont laissées à la charge du nœud. Le listing 4.1 présente le squelette de l'implémentation d'une primitive d'adaptation qui revient principalement à la mise en œuvre d'une commande concrète réversible du patron de conception *Command* [GHJV94].

Listing 4.1 – Définition d'une primitive d'adaptation

```
public class MyConcretePrimitiveCommand extends PrimitiveCommand {
  public boolean execute() {
    // save current state to allow rollback
    // apply command
  }
  public boolean rollback() {
    // rollback to the previous state
  }
}
```

Dans l'exemple, les nœuds sont représentés par les boites grises qui contiennent les composants et sont nommés “node0” et “node1”.

Groupes Un groupe est dédié à la synchronisation de la représentation par modèle d'un ensemble de nœuds. Cette synchronisation définit une portée spécifiée par un protocole de synchronisation, mais également un protocole de communication entre un ensemble de nœuds [FDP+12b]. De la même façon que les *channels* sont utilisées pour définir la sémantique de communication entre les composants, les groupes sont utilisés pour définir la sémantique de communication pour la dissémination du modèle entre les nœuds. La cohérence de l'ensemble du système est donc assurée par les groupes de synchronisation.

Dans l'exemple, le groupe est nommé “*group*” et permet de synchroniser les deux nœuds entre eux. Bien que représenté comme une seule entité et tout comme les canaux de communication, un fragment du groupe est exécuté sur chacun des nœuds afin qu'il puisse communiquer et donc synchroniser la configuration des deux nœuds.

Topologie réseau La topologie réseau est représentée par un ensemble de connexions entre nœuds (*NodeNetwork* et *NodeLinks*) qui définissent les liens réseaux entre les différents nœuds. Chaque lien possède un ensemble de propriétés définissant les caractéristiques du réseau associé. Cette topologie permet ensuite aux *groupes* ou *channels* de récupérer les informations nécessaires pour établir une connexion avec les éléments distants.

Dans la figure de l'exemple, la topologie réseau n'est pas représentée, mais existe tout de même. Ici elle correspond à des adresses réseau qui peuvent être utilisées par les fragments du groupe et du canal de communication pour établir des connexions entre les fragments et ainsi envoyer des données ou synchroniser le modèle.

Patron Type/Instance Pour mettre à jour un système de façon continue, il est important de pouvoir différencier les types représentant les fonctionnalités disponibles ainsi que les paramètres utilisés pour configurer ces fonctionnalités, des instances représentant l'usage localisé de ces fonctionnalités avec un paramétrage spécifique. C'est une nécessité à la fois pour raisonner sur la substituabilité des fonctionnalités mais également pour connaître les instances concernées par une mise à jour de type. L'ensemble des concepts de Kevoree suit donc un patron de conception type/instance [WJ96] qui, à la manière de la programmation objet, sépare les définitions (Classe) des instances (Objet).

Un type est défini par son arbre d'héritage et le type de dictionnaire qui lui est associé (voir figure 4.2). Le paramétrage d'une instance se fait au travers de son dictionnaire qui regroupe l'ensemble des attributs pouvant servir à la configuration du type correspondant. De la même façon que pour son utilisation dans les langages à objets, l'héritage de type permet de mutualiser les fonctionnalités déjà définies dans des *super types* et facilite aussi la spécialisation de ces types en permettant de définir de nouvelles fonctionnalités ou d'en redéfinir certaines. La sémantique d'héritage dans Kevoree est proche de la sémantique d'héritage pour les interfaces Java. Cela permet de spécifier qu'un type va respecter l'ensemble des propriétés, ports, primitives d'adaptations définis dans ses types parents.

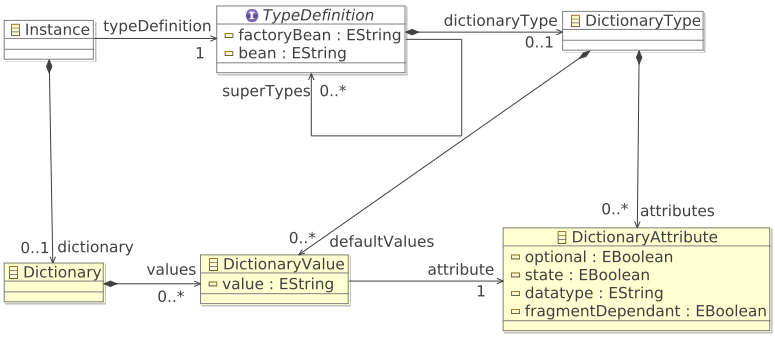


FIGURE 4.2 – Paradigme Type/Instance, dictionnaire et héritage de type

Dans notre exemple, nous pouvons identifier quatre types pour six instances. Tout d'abord le type du groupe qui définit le type de synchronisation entre les configurations des deux nœuds. Ici ce groupe effectue de la dissémination de modèle sur l'ensemble des nœuds connectés (*broadcast*) en utilisant une couche HTTP. Les deux nœuds partagent le même type qui est le type de base dans Kevoree et qui est nommé *JavaSeNode* car il supporte des composants, canaux de communication et groupes développés dans le langage Java. Le canal de communication possède lui aussi son propre type et transmet les messages entre les composants au travers de *socket* réseau Java. Ce canal de communication possède comme propriétés de configuration, dans son dictionnaire, les ports réseau utilisés pour établir la communication entre les deux nœuds. Enfin, les deux composants possèdent le même type qui représente la console servant à émettre et recevoir les messages entre les deux utilisateurs. Cet envoi et cette réception sont modélisés par l'intermédiaire des ports de type *message* de chaque composant. Un port de type message a pour rôle d'envoyer des données sans attendre de retour du récepteur. Il existe aussi des ports de type *service* qui eux permettent d'effectuer des communications synchrones.

4.2.2.2 Sémantique du modèle de configuration

Cycle de vie Afin de permettre leur usage dans un système adaptatif, les instances pouvant être définies dans Kevoree (composants, canaux de communication, nœuds et groupes) possèdent un cycle de vie similaire présenté dans la figure 4.3. Ainsi une instance peut être installée, désinstallée, démarrée et arrêtée (correspond à l'état configuré de la figure). Le passage d'un état à un autre s'effectue par l'intermédiaire des primitives d'adaptation que possède le système. Cependant, chaque type peut définir le comportement de ses instances lors de certaines transitions. Ainsi bien que les opérations d'installation et de désinstallation restent spécifiques à la plate-forme d'exécution, les opérations de démarrage, d'arrêt et de mise à jour (correspond à la transition *adapt* dans la figure) peuvent être spécialisées par chacun des types. Dans notre exemple, le démarrage des consoles est spécialisé pour créer l'interface graphique permettant aux utilisateurs de lire et écrire des messages. Le canal de communication va, quant à lui, démarrer une *socket* réseau pour recevoir les messages provenant des composants distants. De la même façon, le groupe va lui aussi démarrer une *socket* réseau pour pouvoir recevoir les reconfigurations provenant des nœuds distants. Enfin, les nœuds vont initialiser les différents éléments pour pouvoir exécuter les primitives d'adaptation comme charger le module lui permettant de télécharger dynamiquement les librairies de composants que le modèle utilise.

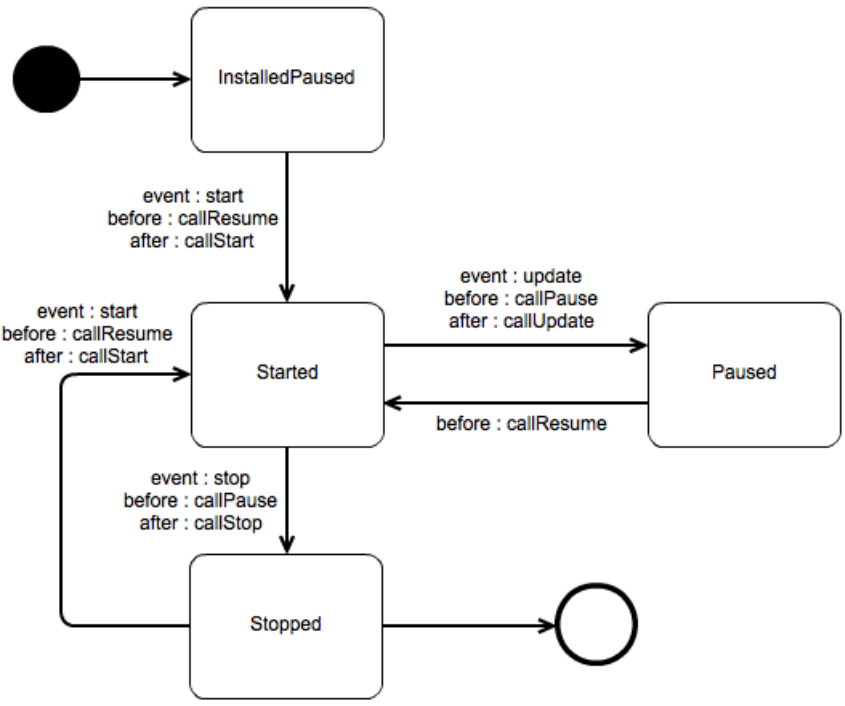


FIGURE 4.3 – Cycle de vie d'une instance

Kevoree et modèle à l'exécution Kevoree est issu des travaux dans le domaine de l'Ingénierie des modèles et plus particulièrement des travaux sur les modèles à l'exécution (voir section 2.2).

Ce processus de M@R est le cœur du système Kevoree et est encapsulé dans un module appelé Kevoree Core qui est embarqué dans chacun des nœuds. Ce module offre à chaque élément du système (nœuds, composants, canaux de communication, groupes) un accès au modèle courant et leur permet aussi de soumettre de nouvelles configurations au travers de nouveaux modèles. Si le Kevoree Core reçoit un nouveau modèle, il a la charge de la validation, puis de la planification de l'adaptation à effectuer et enfin de l'exécution de cette adaptation. L'ensemble de ces étapes est effectué de manière transactionnelle.

La validation du modèle est déléguée à tout élément du système enregistré en tant que *ModelListener*. Chaque composant, canal de communication ou nœud peut déclarer cette interface et s'enregistrer au niveau du *Kevoree Core* chargé de la gestion du modèle. Une fois enregistrée, l'instance sera notifiée concernant les différentes étapes d'une reconfiguration. Pour cela, l'interface *ModelListener* définit les notifications suivantes :

- *preUpdate* permet aux *ModelListener*s d'être notifiés qu'une mise à jour a été proposée. Chaque *ModelListener* peut ainsi valider la configuration proposée.
- *preAllUpdate* permet aux *ModelListener*s d'être notifiés que la mise à jour proposée a été validée par l'ensemble des *ModelListeners*.
- *postUpdate* permet aux *ModelListeners* d'être notifiés que la mise à jour a été appliquée. Chaque *ModelListener* peut ainsi valider que la mise à jour ne pose pas de souci.
- *postAllUpdate* permet aux *ModelListener*s d'être notifiés que la mise à jour qui a été appliquée a aussi été validée par l'ensemble des ModelListeners.
- *preRollback* permet aux *ModelListener*s d'être notifiés que la mise à jour a échoué et qu'un retour à la configuration précédente va être effectué.
- *postRollback* permet au *ModelListener*s d'être notifiés que le retour à la configuration précédente a bien été effectué.

La figure 4.4 représente sous forme de diagramme états-transitions l'intégration des *ModelListeners* avec le processus de *Model@Runtime*. L'état *check* délègue la vérification à l'ensemble des *ModelListeners*. Si l'ensemble des *ModelListeners* accepte le modèle alors le système passe dans l'état *adapt* qui notifie l'ensemble des ModelListeners que le modèle a été validé et qui tente de mettre en place le modèle sur le nœud local. Si la mise en place sur le nœud local réussit alors le système passe dans l'état *validate* qui permet de sauvegarder le modèle comme étant le modèle courant du système et qui notifie l'ensemble des *ModelListeners* afin qu'ils puissent vérifier que la mise à jour a été appliquée. Si l'ensemble des *ModelListeners* valide que l'état courant correspond bien au modèle proposé alors le système passe dans l'état *terminate* qui notifie à tous les *ModelListeners* que la mise à jour a été appliquée avec succès. Outre ces transitions du cas nominal, il y a aussi des transitions pour les cas particuliers. Par exemple, dans l'état *check*, si l'un des *ModelListeners* refuse le modèle alors le système passe dans l'état *rollback* qui sert à revenir en arrière. De la même façon, si le modèle ne peut être appliqué sur le nœud local, le système passe aussi dans l'état *rollback*. Enfin, dans l'état *validate*, l'un des *ModelListeners* ne valide pas que l'état courant correspond bien au modèle proposé, alors le système passe là encore dans l'état *rollback*. Cet état *rollback* permet dans ces différents cas de revenir sur l'ancien modèle, c'est-à-dire d'annuler toutes les modifications qui ont pu être mises en place afin d'appliquer la nouvelle configuration. Pour cela, les *ModelListeners* sont notifiés au début du rollback pour qu'ils puissent eux aussi annuler les opérations qu'ils ont pu commencer. À la fin du *rollback*, les *ModelListeners* sont aussi notifiés pour les informer que la configuration précédente a bien été réappliquée.

Les groupes sont, par défaut, des instances de *ModelListeners*. Contrairement aux composants, aux canaux de communication et aux nœuds qui peuvent implanter l'interface *ModelListener* pour être notifiés des changements locaux, les groupes implantent par défaut cette interface et sont automatiquement enregistrés au niveau du Kevoree Core. Ces instances particulières de

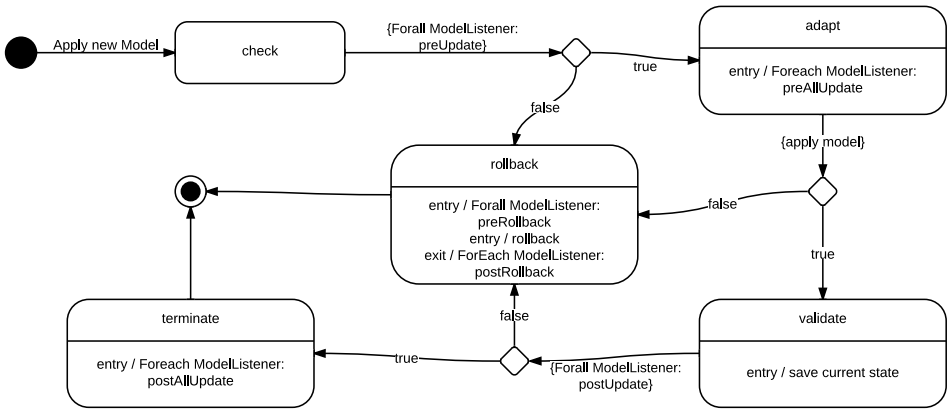


FIGURE 4.4 – Diagramme d'états-transitions montrant l'intégration des ModelListeners dans le processus de Model@Runtime

ModelListener sont utilisées pour la synchronisation et la validation des reconfigurations entre plusieurs nœuds. En effet, chaque nœud héberge un module Kevoree Core lui permettant d'évoluer indépendamment des autres nœuds. Mais les éléments présents sur chacun des nœuds sont capables de modifier l'ensemble du système et si la reconfiguration impacte plusieurs nœuds, il peut être nécessaire que plusieurs nœuds valident le modèle et/ou se synchronisent lors de sa mise en place.

Outre la validation du modèle par l'intermédiaire des *PreUpdate*, l'ensemble des notifications sert aussi à la génération d'événements dans le cadre de la phase d'observation pour pouvoir informer les différents moteurs de décision présents dans le système qu'une mise à jour soit en cours, qu'elle a été effectuée ou qu'elle a échoué.

4.2.3 Support de l'hétérogénéité

Kevoree fournit un ensemble de cadre de développement pour la définition de nouveaux types. Il fournit notamment un cadre de développement pour la conception de type pour microcontrôleur, un cadre de développement pour la conception de type natif (en C et C++), un cadre de développement pour la conception Java utilisable aussi pour la conception sur Android, et un cadre de développement pour la conception JavaScript utilisable au niveau d'un serveur d'applications JavaScript comme NodeJs [3] ou au sein du navigateur.

4.2.4 Extensibilité de Kevoree

Kevoree fournit un ensemble d'implantations pour la définition de systèmes distribués que ce soit au niveau des groupes pour la synchronisation des reconfigurations (Gossip, Paxos, broadcast), que ce soit au niveau des nœuds d'exécution (Java, Android, microcontrôleur, C/C++, Javascript, docker, lxc, jails) ou encore au niveau des canaux de communication (Gossip, de broadcast). En plus de ces implantations, Kevoree permet de définir de nouveaux protocoles

3. http ://www.nodejs.org

de dissémination et de communication, mais aussi la possibilité de définir de nouveaux nœuds d'exécution. C'est cette capacité à pouvoir définir de nouveaux supports d'exécution qui permettent l'extension de Kevoree notamment dans le cadre de son utilisation pour la mise en place de systèmes autonomiques [LMD13]. En effet, les nœuds sont les dépositaires de deux des phases de la boucle autonomique [GC03]. Ces phases, la planification et l'exécution, permettent d'étendre les capacités du système par la définition de nouvelles primitives d'adaptation et par leur intégration dans la construction des plans d'exécution.

4.2.4.1 Délégation de l'exécution de l'adaptation

L'exécution de l'adaptation dans Kevoree est dépendante du support d'exécution et c'est pourquoi, ce sont les types de nœuds qui définissent les implantations des primitives d'adaptation. Mais chaque support d'exécution peut avoir ses propres capacités d'adaptation. En effet, si l'on considère un nœud Java permettant d'exécuter un ensemble de composants dans une machine virtuelle Java, ces capacités d'adaptation se cantonnent à la manipulation de composants et de canaux de communication alors qu'un nœud capable de gérer une infrastructure de Cloud doit par exemple être capable de manipuler non seulement des composants chargés de son administration, mais aussi des nœuds représentant les machines virtuelles correspondant aux plates-formes.

C'est pourquoi le processus de Model@Runtime implanté dans Kevoree délègue aux nœuds, la déclaration de l'ensemble des capacités d'adaptation qu'ils fournissent (voir figure 4.5).

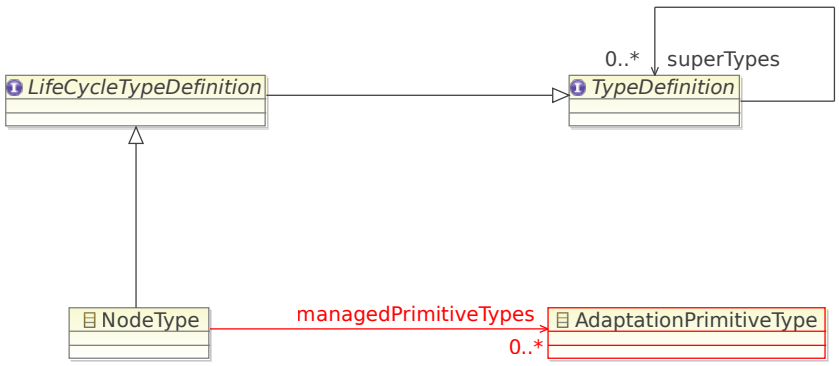


FIGURE 4.5 – Méta-modèle Kevoree avec la représentation des primitives d'adaptation

Ainsi en plus des propriétés de son dictionnaire, un type de nœud spécifie ses primitives d'adaptation. Dans Kevoree, le nœud par défaut appelé *JavaSeNode* est capable de manipuler les composants et canaux de communication ainsi que les groupes de synchronisation. Ces primitives définissent de manière abstraite les capacités d'adaptation disponibles sur un type de nœud.

4.2.4.2 Délégation de la planification de l'adaptation

Comme les actions d'adaptation sont dépendantes du support d'exécution, seul le nœud est capable d'identifier quelles actions permettent de passer d'un modèle à un autre. C'est pourquoi, Kevoree délègue la comparaison de modèle au nœud. Cette comparaison fait partie de la phase de planification définie dans le modèle MAPE [GC03, LMD13].

Mais la phase de planification ne se limite pas seulement aux choix des actions nécessaires pour appliquer une adaptation. Elle consiste aussi à ordonnancer les actions. En effet, à partir du modèle de la figure 4.6 dans lequel un composant a de type A envoie des données à un canal de communication y de type Y et ce canal de communication transmet ces données à un composant b de type B, la mise en place de ce modèle correspond aux actions suivantes : *installer le type A*, *installer le type B*, *installer le type Y*, *installer l'instance a*, *installer l'instance b*, *installer l'instance y*, *connecter a avec y*, *connecter y avec b*, *démarrer l'instance a*, *démarrer l'instance b*, *démarrer l'instance y*.

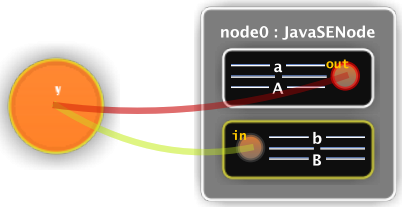


FIGURE 4.6 – Exemple de configuration nécessitant une planification

Ici, l'ordre d'exécution de cet ensemble d'actions a une importance, car si a démarre avant y et qu'il envoie des données, celles-ci ne seront peut-être pas reçues si y n'est pas démarré et de manière identique, il faut que b soit démarré avant y.

Tout comme la comparaison de modèle est spécifique au type de nœud, l'ordonnancement des actions est, lui aussi, spécifique. En effet, les dépendances entre les actions dépendent fortement des implantations des actions qui elles-mêmes sont spécifiques au type de nœud. C'est pourquoi le processus du Kevoree Core, qui délègue la comparaison et l'exécution au nœud, délègue aussi l'ordonnancement.

La définition d'un type de nœud se traduit donc par la spécification des primitives d'adaptation, du dictionnaire, des méthodes de cycle de vie et enfin les méthodes de planification et d'exécution. La définition de ces différents éléments est à la charge des experts d'une plate-forme particulière.

Ce modèle de configuration devient le quatrième point de vue de l'approche permettant de modéliser et administrer des systèmes adaptatifs hétérogènes et distribués. La validation de ce modèle n'est pas effectuée dans cette partie mais présentée en détail dans la partie suivante. La présentation de ces différentes contributions permet de faire apparaître une approche pour la modélisation de la configuration de ces systèmes centrée autour de quatre points de vue intégrés (voir chapitre 2 et plus précisément la section 2.2.1) :

- *un point de vue de configuration : Kevoree (chapitre 4),*
- *un point de vue pour la modélisation de la variabilité, fondé sur CVL, fournissant une modélisation des potentiels composition de fragments d'architecture (section 3.3),*
- *un point de vue spécifiant les actions d'adaptation disponibles. Ce point de vue est défini à l'aide de SmartAdapters fournissant des opérateurs de composition de haut niveau*[4] *) (section 3.2).*

4. une nouvelle mise en œuvre a été proposée plus récemment avec une approche de pattern développée par la société Thales

— *un point de vue de modélisation du contexte générique, sur lequel nous intégrons des sondes logicielles, à partir d'une approche de composition supportant le code patrimonial (section 3.2).*

Cette vision est rendu réaliste et maintenable par l'amélioration des techniques de modélisation aussi bien pour leur utilisation à l'exécution (section 3.4) que par leur adaptation à l'hypothèse d'un monde plus ouvert dans lequel les différents méta-modèles proposés sont amenés à évoluer et dans lequel de nouveaux points de vue peuvent être intégrés (section 3.1).

Deuxième partie

Expérimentations dans le cadre de ces recherches

Afin d'insister dans ce document de synthèse sur l'importance de l'expérimentation dans ma démarche scientifique, cette partie présente un résumé non exhaustif, mais représentatif, de différentes expériences menées dans le cadre des thèses que j'encadre ou que j'ai co-encadrées afin de montrer la pertinence des approches de modélisation à l'exécution et des opérateurs de composition de modèles associés.

Afin de valider les différentes contributions de ce travail de recherche, nous avons cherché à confronter différents cas d'usage dans lesquels une approche de modélisation à l'exécution a pleinement du sens face à une problématique de configuration. Nous avons appliqué ces techniques à quatre domaines pour lesquels la logique de configuration est complexe :

- le domaine de l'environnement mobile,
- le domaine du *cloud computing*,
- le domaine de l'Internet des Objets,
- le domaine de la surveillance de la consommation de ressources en environnements ouverts.

Pour chacun de ces domaines correspondant à un chapitre pouvant être lu de manière indépendante dans cette partie de mémoire, nous rappelons le contexte et l'objectif de l'étude correspondant généralement à une des questions de recherche suivantes.

- **RQ4.** Comment maintenir la cohérence de ce modèle représentant l'état du système dans un environnement distribué ? Ce point reste un challenge particulièrement dans des environnements mobiles où les communications sont intermittentes ? *Cette question est tout particulièrement abordée dans le chapitre 5.*
- **RQ5.** Comment valider la pertinence de l'abstraction proposée et sa pertinence pour le domaine visé ? *Cette question est tout particulièrement abordée dans les chapitres 6 et 7.*
- **RQ6.** Comment réutiliser les approches, méthodes et outils habituellement utilisés sur un modèle de conception, à l'exécution ?
- **RQ7.** Comment adapter les approches, méthodes et outils de modélisation à l'hypothèse du monde ouvert ? *Ces deux dernières questions sont particulièrement abordées dans les chapitres 8 et 9.*

Les trois premières questions ont plutôt été adressées au travers de la proposition autour de i) l'utilisation des modèles à l'exécution afin d'utiliser entres autres les approches de modélisation de la variabilité et les approches de composition de modèles à l'exécution, ii) la construction d'un langage de configuration suivant cette approche pour la classe des systèmes dynamiques hétérogènes et adaptables.

Chapitre 5

Models@Runtime en environnement mobile

Sommaire

5.1 Exemple de mise en œuvre d'un nouveau type de groupe : Gossip **61**
5.1.1 Objectifs et spécificité d'une dissémination selon un modèle pair à pair 61
5.1.2 Une architecture moyenne calculée comme une agrégation épidémique *gossip* 61
5.1.3 Principe de combinaison des horloges vectorielles ainsi qu'une propagation *gossip* 62
5.1.4 Protocole *gossip* pour dissémination de *Model@Runtime* 62
5.1.5 Propriétés attendues 65
5.1.6 *Slicing* de modèle à l'image du *peer sampling* 67
5.2 Validation expérimentale sur *cluster* de simulation **68**
5.3 Protocole expérimental commun **68**
5.3.1 Modèle de topologie initiale 69
5.3.2 Horloge de temps absolu pour la collecte des traces d'exécution . . 69
5.3.3 Mode de communication 69
5.4 Expérimentation 1 : Délai de propagation vis-à-vis de l'usage du réseau de communication **70**
5.4.1 Protocole expérimental 70
5.4.2 Limites de validité expérimentale 70
5.4.3 Analyse des résultats expérimentaux 71
5.5 Expérimentation 2 : Impact des erreurs de communication sur les délais de propagation **73**
5.5.1 Protocole expérimental 73
5.5.2 Limites de validité expérimentale 74
5.5.3 Analyse des résultats expérimentaux 74
5.6 Expérimentation 3 : Réconciliation de modèle et reconfigurations concurrentes **74**
5.6.1 Protocole expérimental 75
5.6.2 Limites de validité expérimentale 75
5.6.3 Analyse des résultats expérimentaux 75
5.7 Conclusion sur l'usage des groupes pour la convergence **76**

Un des points important et original du langage de configuration présent dans Kevoree est la réification, comme un élément de premier niveau, de la notion de groupe permettant d'encapsuler la ou les sémantique(s) de dissémination d'un modèle de configuration dans un environnement distribué. Ce chapitre est une expérimentation menée dans le cadre de la thèse de François Fouquet mais avec une contribution significative d'Erwan Daubert afin d'évaluer la pertinence de ce modèle de groupe et sa capacité à capturer une sémantique de dissémination complexe dans le cadre d'environnements mobiles.

Le première axe d'évaluation critique pour l'utilisation de modèles à l'exécution dans le cadre de systèmes dynamiques, hétérogènes et distribués est d'étudier le sens de cette notion de modèle de configuration dans un environnement hautement distribué souvent caractérisé par l'absence d'un état global. Plus précisément, il est nécessaire de comprendre comment capturer la sémantique de synchronisation pour cette couche de réflexion distribuée et comment synchroniser plusieurs nœuds (conteneur d'exécution) dont chaque modèle de configuration peut évoluer de manière indépendante. Si des projets comme Zookeeper d'apache [1] sont des systèmes de coordination hautement disponibles, permettant à des développeurs d'écrire des programmes qui pourront une fois déployés sur un réseau coordonner leurs actions tout en tolérant des éventuelles pannes, l'objectif dans Kevoree au travers la notion de groupe est de fournir une abstraction pour ces problématiques de configurations distribuées tout en tolérant au sein d'un même système plusieurs sémantiques de coordination et de dissémination du modèle de configuration.

De ce fait, nous cherchons au travers d'une expérimentation placée dans un cadre de réseau maillé à la topologie très dynamique pour une complexité de systèmes importante liée au nombre de nœuds à coordonner, à valider deux points :

— la capacité de Kevoree à encapsuler un algorithme de convergence afin de faire converger les copies de modèles des différents nœuds ;
— la capacité de Kevoree à encapsuler un algorithme de dissémination qui permet la dissémination dans un *cluster* large échelle.

Ce chapitre traite de l'évaluation de l'implantation d'un type de groupe de type Gossip Kevoree alliant une inondation *gossip* et une traçabilité par *VectorClock* détaillée en 5.1 vis-à-vis de ces besoins. En effet l'inondation par *gossip* est choisie pour résister à un réseau maillé à la topologie très dynamique tandis que les *VectorClock* permettent la traçabilité des modifications à la fois dans ce réseau et dans les modèles de configuration que l'on souhaite disséminer. Par cette traçabilité, nous pouvons alors déterminer l'existence de sous-groupes isolés déterminant seuls de nouvelles configurations et divergeant du modèle de configuration commun. L'approche est donc nettement inverse d'une implémentation de groupes consensus (par exemple un groupe implantant un algorithme de type Paxos), puisqu'ici le but est de laisser les nœuds diverger pour résister à la perte de connexion et continuer d'offrir un service sur le terrain. La validation présentée ci-dessous exploite donc une implantation en Scala et Java des concepts détaillés en 5.1 sur un *cluster* de machines servant à simuler l'environnement mobile sapeur-pompier.

1. xxx

5.1 Description de la mise en œuvre du type de groupe gossip pour les réseaux P2P : association de Gossip et VectorClock

5.1.1 Objectifs et spécificité d'une dissémination selon un modèle pair à pair

Dans ce cas d'usage de réseaux maillés de grande taille, dynamiques, l'hypothèse de connexion persistante n'est plus valide. En outre, pour de nombreux scénarios applicatifs, l'impossibilité de déclencher des reconfigurations dans un système complexe suite à de nombreuses défaillances de nœud n'est pas une option possible.

Pour ce cas d'usage il est important de disposer d'une propagation plus épidémique et plus opportuniste telle que celle proposée par les protocoles *gossip*. Cette pandémie peut venir de n'importe quels nœuds puisque chacun peut avoir la liberté d'héberger des raisonneurs capables de modifier potentiellement le modèle de configuration.

Cette sous-section détaille l'usage de protocole *gossip* en tant que groupe Kevoree et donc dédié à la propagation épidémique de modèle. Ainsi dans cette approche chaque fragment d'un groupe *gossip* est dédié à propager de façon épidémique ses modifications aux autres fragments. Pour cela il va échanger ce modèle avec un de ses voisins, qui fera alors de même de proche en proche jusqu'à la convergence du *cluster*.

5.1.2 Une architecture moyenne calculée comme une agrégation épidémique *gossip*

Dans ce mode de propagation aucun ordre n'est assuré pour les mises à jour, et aucun verrou n'est posé. Un problème intervient cependant, lorsque deux nœuds calculent deux modèles différents à chaque extrémité du réseau P2P : ces modèles sont propagés de façon concurrente. De même lorsque qu'un réseau P2P possède des sections isolées pour cause de perte de communication, chaque sous-réseau va évoluer et donc faire émerger son propre modèle. Lorsque ces modèles concurrents entrent en collision il faut alors prendre une décision pour la réconciliation de ces informations.

Dans l'article [JMB05], Jelasity *et al.* proposent une agrégation épidémique fondée sur un protocole *gossip*. Le principe est le suivant : chaque nœud a besoin d'une valeur agrégée du *cluster* pour prendre des décisions mais l'appel à tous les nœuds est trop coûteux. A l'inverse, chaque nœud fait une moyenne avec ses voisins à l'aide d'échange *gossip* pour maintenir une valeur la plus proche possible de celle contenue en pratique dans le *cluster*. Dans l'article [JB06a], les auteurs ont appliqué le même principe dans le projet *T-man* pour calculer une topologie de façon empirique.

Le besoin de valeur pondérée est transposable aux modèles structurels. En effet, lorsque deux modèles sont concurrents , il est possible de réaliser une opération de fusion afin de produire un modèle agrégeant les données des deux parties. À la manière de l'agrégation épidémique proposée par Jelasity, la contribution de ce groupe *gossip* est d'assurer un modèle de configuration moyen. Cependant à la différence d'une valeur chiffrée, les modèles doivent détailler d'avantage d'information sur leur provenance, afin de détecter les conflits et pouvoir faire la réconciliation à la manière d'un gestionnaire de version de code source tel que *git* [2].

2. http ://git-scm.com

5.1.3 Principe de combinaison des horloges vectorielles ainsi qu'une propagation *gossip*

Le principe général de la solution proposée dans ce *type de groupe gossip* Kevoree est de combiner une propagation à la *gossip* avec des méta-informations pour être capable de détecter les branches divergentes et les réconcilier. Ainsi à la manière d'un gestionnaire de version, les modèles échangés doivent être équipés d'horloges logiques pour détecter ces branches. Cette solution a été publiée à la conférence DAIS'2012 [FDP+12b].

Les horloges logiques sont une solution très utilisée pour ordonner des échanges asynchrones non ordonnés. Lamport *et al.* [Lam78] proposent dans un article de 1978 d'ajouter un temps logique à chaque envoi de message, ce temps logique doit être partagé par l'ensemble des nœuds *via* une phase d'initialisation. Peu après en 1988, Fidge *et al.* [Fid88] et Mattern [Mat89] proposent conjointement d'exploiter un vecteur de temps logique associé à chaque évènement envoyé. Plus décentralisée cette solution n'impose pas d'horloge logique commune puisque l'horloge vectorielle embarquée dans chaque message contient les temps logiques des horloges de chaque nœud traversé par le message. L'horloge vectorielle sert alors à la traçabilité du message et surtout à sa synchronisation sur chaque nœud. Cette solution a été très utilisée par la suite pour gérer des ordres partiels d'évènements [BR02] ou même des synchronisations de base de données *clé/valeur* distribuées, tel que le projet Voldemort [3] de LinkedIn.

5.1.4 Protocole *gossip* pour dissémination de *Model@Runtime*

Le protocole *gossip* de ce groupe échange donc des *VectorClocks* ainsi que des modèles accompagnés de *VectorClock* . Le listing 5.1 définit en Scala ces trois structures, l'horloge logique est représentée par un *Long* , un *VectorClock* est donc essentiellement une liste de *ClockEntry* associant un nœud à une horloge. Le *nodeID* identifie ici le fragment du groupe courant déployé sur le nœud dont le nom correspond à ce *nodeID* .

Listing 5.1 – Définition des VectorClocks

```
case class ClockEntry(nodeID:String,version:Long)
case class VectorClock(entries:List[ClockEntry])
case class VectorClockModel(model:KevModel,vectorClock:VectorClock)
```

Les modèles d'architecture représentant un volume de données important, la propagation *gossip* naïve de cette donnée coûterait beaucoup trop cher en terme d'occupation réseau. La solution proposée vise donc à découper le protocole en deux phases, une phase de diffusion des *VectorClocks* de chaque nœud correspond à un modèle, et un échange entre deux lorsque l'échange de modèle est nécessaire. Voici les grandes lignes de cette solution.

- Diffusion des *VectorClocks* en associant une approche pro-active et réactive. Les nœuds se synchronisent de façon périodique avec leur voisin mais exploitent également un mécanisme pour notifier leurs voisins en cas de modifications. L'association des deux mécanismes permet à la fois de supporter un fort taux d'échec de communication tout en garantissant un temps de propagation court pour les nœuds disponibles sans erreur.
- Communication inversée de la donnée à transférer (modèles) suivant le principe d'Hollywood [Fow04] et donc avec une inversion de contrôle. Là encore ceci est fait pour minimiser les échanges réseaux, les modèles ne sont jamais envoyés spontanément sur le réseau mais toujours demandés activement par un tiers.
- Le *gossip* est alimenté par ce qu'il dissémine (sa charge utile), c'est-à-dire le modèle. En effet les services essentiels que sont le *PeerSampling* et le *PeerSelector* se font en

3. http://project-voldemort.com

exploitant la couche de réflexion du modèle.

— Le service de *PeerSelector* exploite un historique d'état des communications en supplément de la topologie réflexive du modèle afin de réduire les tentatives de communication vers les nœuds en faute et ainsi réduire leur impact sur le fonctionnement du système.

Algorithm Part 1 DEFINITIONS

```
Message ASK_VECTORCLOCK, ASK_MODEL, NOTIFICATION
Type VectorClockEntry := <id : String, version ∈ ℕ>
Type Node // represents a node on the system
Type Model // represents a configuration of the system
Set Group := {node : Node}
Set IDS(g : Group) := {id : String | ∃ node : Node, node ∈ g & node.name = id}
Set Neighbors(originator : Node, g :Group) := {node : Node | node ∈ g & originator ∈ g}
Set VectorClock(originator : Node, g : Group) := {entry : VectorClockEntry | entry.id == originator.name}
 1: ⋃ {entry1 : VectorClockEntry |∃ node : Node, node != originator & entry1.id ∈ IDS(g) & node ∈ g}
Set VectorClocks(originator : Node, g : Group) := {vectorClock : VectorClock(originator, g)}
```

Un protocole *gossip* est dédié à la dissémination d'un état. Chaque fragment d'un groupe *gossip* est donc un participant à ce protocole et stocke donc un état qui lui est propre. Cet état est majoritairement contenu dans chacun des nœud Kevoree avec l'accès au modèle courant mais nécessite également des informations supplémentaires (voir partie algorithme 2). Chaque fragment stocke donc son *VectorClock* courant, un score associé à chacun de ses voisins pour les besoins du service de *PeerSelection* .

5.1.4.1 Algorithme principal (voir partie algorithme 3)

Le protocole proposé est asynchrone et sans état, cependant on peut tout de même parler de cycle pour plus de compréhension. Un cycle *gossip* correspond à l'envoi d'une requête d'état vers un voisin, une comparaison de cet état avec le local et une synchronisation si nécessaire avec le nœud voisin.

Comme tous les *ModelListener* les fragments de groupe *gossip* sont notifiés à chaque changement local du modèle courant, une notification est donc envoyée à tous les voisins directs pour déclencher un cycle *gossip* anticipé chez ces voisins. En retour en début de cycle de protocole les autres fragments vont envoyer un message de requête d'état (structure *ASK_ VECTORCLOCK*) au fragment origine de la modification. Comme les connexions sont volatiles et non sûres, des notifications peuvent être perdues et non reçues par des membres du groupe. Pour faire face à ces pertes, chaque fragment déclenche également un cycle du protocole périodiquement en choisissant alors un voisin cible de la synchronisation *via* la méthode *SelectPeer*. Les modèles étant une donnée d'un volume conséquent, le protocole fait d'abord une demande du *VectorClock* seul puis une demande du *VectorClock* , accompagné du modèle si la synchronisation est nécessaire.

Pour déterminer si la synchronisation est nécessaire, le protocole repose sur une comparaison des *VectorClocks* . De façon très simplifiée cette comparaison évalue deux à deux les entrées des *VectorClocks* . Si un nouveau nœud est présent dans le *VectorClock* distant, ou si une version d'une de ses entrées est supérieure au local il est déclaré comme étant postérieur. A l'inverse

Algorithm Part 2 STATE

```
1: localFragment : Group // pointer to local group instance fragment
2: currentModel : KevModel // local version of system configuration
3: localNode : Node // representation of local node instance
4: currentVectorClock ∈ VectorClocks(localNode, g)
5: scores := {<node : Node, score>, node ∈ Neighbors(localNode, g) && score ∈ ℕ}
6: nbFailure := {<node : Node, nbFail>, node ∈ Neighbors(localNode, g) && nbFail ∈ ℕ}
```

si un nœud est manquant dans le distant ou si une version est inférieure il est déclaré comme antérieur. Si toutes les versions ne sont pas strictement égales ou supérieures le *VectorClock* distant est déclaré comme concurrent.

À la réception d'un *VectorClock* , un fragment effectue donc la comparaison avec sa copie locale. Si le *VectorClock* reçu est antérieur il est simplement ignoré et le cycle est fini. S'il est postérieur ou concurrent, une demande de *ASK_MODEL* est envoyée. Lorsque qu'un modèle accompagné d'un *VectorClock* est reçu par un fragment il refait une comparaison, et si le modèle est toujours postérieur il fait la mise à jour avec ce nouveau modèle. Si les deux modèles sont toujours concurrents cela signifie que le *VectorClock* local et le distant ont des modifications concurrentes et proviennent donc d'une divergence. Afin de garantir la pérennité des modifications le fragment local fait une fusion (*merge*) des *VectorClocks* ainsi que du modèle avant de faire la mise à jour locale. La fusion des modèles peut être soumise à des conflits qui peuvent être résolus à l'aide de règles de priorité suivant les cas métiers (par exemple en donnant la priorité à la donnée passée par des nœuds maîtres).

L'algorithme 3 décrit le pseudo-code de ce protocole.

Algorithm Part 3 ALGORITHM

```
On init() :
 1: vectorClock ← (localNode.name, 1)
 2: scores ← {Neighbors(localNode, g) × {0}}
On change (currentModel) :
 3: incrementLocalVectorClock()
 4: ∀ n, n ∈ Neighbors(localNode, g) → send (n, NOTIFICATION)
Periodically do() :
 5: node ← selectPeerUsingScore()
 6: send (node,ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode, g), NOTIFICATION) :
 7: send (neighbor, ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode,g), remoteVectorClock ∈ VectorClocks(neightbor, g)) :
 8: result ← compareWithLocalVectorClock (remoteVectorClock)
 9: if result == BEFORE || result == CONCURRENTLY then
10:     send (neighbor, ASK_MODEL)
11: end if
On receive (neightbor ∈ Neighbors(localNode,g),vectorClock ∈ Vectorclocks(neighbor, g), model)) :
12: result ← compareWithLocalVectorClock (targetVectorClock)
13: if result == BEFORE then
14:     updateModel(model)
15:     mergeWithLocalVectorClock(vectorClock)
16: else if result == CONCURRENTLY then
17:     resolveConcurrently(vectorClock, model)
18: end if
On receive (neighbor ∈ Neighbors(localNode,g), request) :
19: if request == ASK_VECTORCLOCK then
20:     send (neighbor, currentVectorClock)
21: end if
22: if request == ASK_MODEL then
23:     send (neighbor, <currentVectorClock,currentmodel>)
24: end if
```

5.1.4.2 Fonction *SelectPeer* (voir Algorithme 4)

La fonction *SelectPeer* est appelée périodiquement pour choisir un nœud pour lancer un cycle de *gossip* et se synchroniser avec lui. Ce service est essentiel pour assurer la qualité de la distribution uniforme des échanges, qui idéalement doivent s'organiser suivant une loi aléatoire [JVG+07]. Différentes implantations de ce service sont disponibles dans la littérature avec différentes propriétés de convergence. Celle proposée ici est donc bien évidement interchangeable avec une approche plus classique de *peer sampling* suivant une loi aléatoire.

La fonction présentée ici cherche une répartition uniforme dans le temps des nœuds joignables tout en réduisant l'impact des nœuds en faute en limitant leur taux de synchronisation.

Les liens avec les nœuds fils sont obtenus par le modèle de configuration. Pour capturer cette information, le modèle Kevoree possède une notion de *NodeLink* orienté. Ces *NodeLinks* forment un graphe orienté qui répond aux besoins de topologie pour les réseaux P2P non uniformes comme expliqué par Kermarrec *et al.* [KvS07].

En cas d'échec lors de la sélection d'un nœud, son score augmente de deux fois son taux d'échec précédent. Le score d'un nœud est remis au minimum des scores des nœuds lorsqu'une communication est établie après un échec. Le score d'un nœud augmente de un lorsqu'une synchronisation est effectuée. Ce mécanisme laisse donc une chance aux nœuds non joignables de revenir dans le processus mais rend prioritaires les nœuds joignables donc la dernière synchronisation est plus ancienne que les autres. Le listing 4 détaille cette fonction.

Algorithm Part 4 SelectPeer

```
Function selectPeerUsingScore()
 1: minScore := ∞ ; potentialPeers := {}
 2: for node → Neighbor(localNode, g) do
 3:     if node != localNode && getScore(node) < minScore then
 4:         minScore := getScore(node)
 5:     end if
 6: end for
 7: for node → Neighbor(localNode, g) do
 8:     if node != localNode && getScore(node) == minScore then
 9:         potentialPeers := potentialPeers ⋃ {node}
10:     end if
11: end for
12: node := select randomly a node from potentialPeers
13: updateScore(node)
14: return node
Function getScore(node ∈ Neighbors(localNode, g))
15: return scores(node)
Function updateScore(node ∈ Neighbors(localNode, g))
16: oldScore := getScore(node)
17: scores := scores ⋃ {node, oldScore + 2 * (nbFailure + 1)} \ {node, oldScore}
```

Le *merge* de *VectorClock* garde les plus grandes valeurs de chaque entrée comme décrit par Fidge et Mattern [Fid88],[Mat89]. La fonction d'incrément du *VectorClock* local est détaillée dans l'algorithme 5.

Algorithm Part 5 FUNCTIONS

```
Function incrementVectorClock()
 1: if changed == true then
 2:     ∀ entry, entry ∈ currentVectorClock & entry.id == localNode.name ⇒ entry.v ← entry.v + 1
 3:     changed ← false
 4: end if
```

5.1.5 Propriétés attendues

L'algorithme proposé est issu d'une composition d'approche de diffusion *gossip* dont le mode de communication est inversé et dont les données échangées sont horodatées à l'aide de *VectorClock* afin de faire converger le système. De cet assemblage sont attendues plusieurs propriétés détaillées dans les sous-sections suivantes. L'objectif de ces propriétés est de borner l'algorithme précédemment défini en terme de complexité temporelle et quantitative en nombre de messages. La validation présentée en section 5 illustre les performances de cet algorithme de façon expérimentale.

5.1.5.1 Propriétés de convergence

Propriété 1 *Si deux nœuds produisent deux modèles d'architecture incompatibles alors au bout d'un temps fini un modèle compatible est produit et diffusé à l'ensemble des nœuds du système.*

La définition précédente de cohérence des modèles en chaque nœud est assurée par deux mécanismes :

- en premier lieu, par la construction d'un ordre partiel distribué sur les modèles, qui sont étiquetés par une horloge vectorielle : un modèle m_2 dépendant causalement d'un autre plus ancien m_1 est compatible avec m_1 ;
- en second lieu, en cas d'absence d'ordre causal entre deux modèles, par la résolution de conflits assurée par un algorithme de calcul différentiel de modèles produisant un modèle compatible.

5.1.5.2 Complexité temporelle

La propagation d'un nouveau modèle depuis un nœud n vers tous les autres est proportionnelle au diamètre D du graphe constitué par le groupe, et au nombre moyen de voisins V dans ce graphe. En effet l'application d'un nouveau modèle au nœud n provoque l'envoi de messages de notification aux voisins de n, qui en réponse lui demandent son horloge vectorielle, constatent qu'elle est supérieure à leur propre horloge, demandent le modèle de n, mettent à jour leur modèle puis procèdent comme n. Il y a donc 5 échanges de messages entre voisins, et l'ensemble des nœuds est donc mis à jour en $(D-1)$ étapes. En l'absence de notification l'envoi de demande de *VectorClock* se fait périodiquement après un délai T établi comme propriété du groupe et donc partagé par toutes les répliques de l'algorithme proposé. Le pire cas temporel de cet algorithme correspond à la configuration sans notification et est donc proportionnel à la valeur périodique et au diamètre du graphe de nœud, la synchronisation se fait alors en 4 étapes seulement. Soit T_M le délai moyen d'envoi d'un message, dans le pire cas des cas le temps total de mise à jour est inférieur ou égal à $(D-1) * V * T * (4 * T_M)$. Le meilleur cas correspond à une sélection immédiate du voisin, ce qui donne une borne inférieure de $(D-1) * T * (4 * T_M)$ sans emploi de notifications. Dans le cas où les notifications sont employées la borne inférieure devient $5 * T_M$.

Les résultats présentés en section 5.4 montrent une évaluation expérimentale de cette complexité sur un cas d'usage de topologie mobile. En pratique également la désactivation des notifications et donc le mode *pull* seul n'est pas une solution satisfaisante. Cependant à la manière du protocole T-Man [JB06a], il est envisageable d'envoyer des notifications à un nombre contrôlé de noeuds, garantissant la non-inondation du réseau , tout en garantissant la convergence. Cet envoi de notification de manière non aveugle n'est pas détaillé ici mais peut largement profiter de l'approche Models@Runtime et de son historique pour par exemple sélectionner les nœuds les plus régulièrement mis à jour.

5.1.5.3 Propriétés de résilience à l'intermittence des connexions

Propriété 2 *Toute coupure de communication entre deux nœuds voisins entraîne une forte probabilité de création d'un sous-réseau et donc d'une divergence de deux modèles évoluant de manière incompatible. Plus la résolution est faite rapidement plus elle est considérée comme simple.*

Les évènements de reconnexion sont exploités afin d'améliorer cette propriété.

- Dès sa reconnexion un nœud déclenche un cycle de *gossip* avec le voisin dont la dernière synchronisation est la plus ancienne.

— À l'inverse toute reconnexion d'un voisin précédemment en erreur entraîne une augmentation de sa priorité pour le prochain cycle. En revanche, si aucune reconnexion n'est observée le noeud est progressivement éliminé de la sélection.

Une validation expérimentale de cette propriété est proposée en section 5.6.

5.1.5.4 Complexité en nombre de messages

Propriété 3 *Les échanges de modèles sont considérés comme plus coûteux que les envois de notification et de* VectorClock .

Soit N le nombre de nœuds du graphe liés au groupe considéré, et soit V le nombre moyen de voisins. Chaque nœud va chercher une stabilisation avec ses voisins, ce qui va nécessiter à chaque étape une notification et un échange de *VectorClock* . À cela s'ajoute un envoi de modèle lorsque la réception d'une horloge vectorielle montre qu'une synchronisation est nécessaire. Une borne supérieure du nombre de messages est la suivante :

$$(N-1)*V*NOTIFICATION + ASK_VECTOR_CLOCK + VECTOR_CLOCK$$

$$+(N-1)*(ASK_MODEL + MODEL)$$

Dans le cas où le système de notification est employé, V prend la valeur 1, ce qui correspond également au meilleur cas sans notification. Que ce soit avec ou sans notification, l'envoi du modèle vers tous les noeuds est inévitable. L'usage d'un envoi en deux passes (horloges puis modèles) permet ainsi de réduire l'envoi de modèle sans raison, l'impact de ce choix sera évalué de façon expérimentale dans la section 5.4.

5.1.6 *Slicing* de modèle à l'image du *peer sampling*

Les propriétés décentralisées du *gossip* ont permis et amené son utilisation sur des réseaux P2P fédérant un très grand nombre de nœuds. La construction exhaustive de la topologie est déclarée comme non envisageable dans la littérature. Le *slicing* et le *peer sampling* sont alors la solution proposée pour extraire localement la liste utile et nécessaire des nœuds voisins pour les échanges *gossip*. La topologie est alors répartie en *slice* (sous-partie) sur différents nœuds.

La taille de la topologie due au nombre de nœuds n'est pas le seul problème, d'ailleurs dans le cas d'étude sapeur-pompier, les groupes n'atteignent pas une telle masse critique. Comme le rappelle Jelasity *et al.* dans un article sur le protocole *T-Man* [JB06a] et dans [JVG+07] le problème n'est pas tant la taille de stockage de la topologie mais plutôt sa maintenance dans un système fédérant des nœuds très sporadiques qui se met perpétuellement à jour. En effet la mobilité et les connexions/déconnexions rapides des nœuds modifient la topologie de façon très rapide, de sorte que la maintenance globale est très difficile voire illusoire. Ce constat rejoint d'ailleurs celui fait pour les *Ultra-Large-Scale Systems* [NFG+06]. Une réponse à ce problème est la construction incrémentale proposée par le protocole *T-man*. Celui-ci construit la topologie de proche en proche tout en gardant sa visibilité dans des *slices* de nœuds. Ce type d'approche permet à la fois de découper la topologie en sous-parties mais également d'apporter une propriété de *self healing* ‘ celle-ci en auto-réparant les liens non valides.

De manière fonctionnelle, le *slicing* et le *peer sampling* visent à fournir un sous-ensemble *minimaliste et nécessaire* pour la coordination d'une fonctionnalité. Cette fonctionnalité peut être par exemple un service de *SelectPeer* en assurant une répartition uniforme et aléatoire du nombre de synchronisation d'un nœud avec ses voisins. On peut alors faire le parallèle avec le modèle Kevoree qui dans notre approche sert de base topologique pour le *gossip*. Le *slicing* de modèle consiste également à le découper en sous-parties *minimalistes et nécessaires*. La portée

des *slices* dépend de la fonctionnalité à garantir. Pour l'usage en tant que topologie du modèle Kevoree le besoin en connaissance topologique suit les mêmes règles que le *peer sampling*. La topologie Kevoree étant modélisée comme un graphe orienté tout comme les topologies exploitées en *gossip* [KvS07], les deux approches partagent les mêmes opérateurs de découpage.

Le concept *slicing* est lui même complètement cohérent avec le principe même du M@R. En effet le *slicing* est fait pour gérer la divergence de la topologie, tout comme le M@R qui lui cherche à gérer la divergence du modèle de configuration. La capacité du *gossip* à fonctionner malgré cette divergence en fait un choix idéal pour la propagation du M@R.

La portée du *slicing* pour la partie modèle à composant de Kevoree est cependant plus délicate. La visibilité d'un *slice* dépend du besoin des nœuds en réflexion vis-à-vis des autres nœuds du cluster. Par exemple si des nœuds A et B ne sont pas adaptables par des nœuds B et C alors il n'est pas nécessaire de leurs transmettre la partie descriptive des composants de A et B. Le groupe *gossip* injecté par exemple entre ces nœuds peut alors couper le modèle (topologie et modèle de composant) pour n'envoyer que la partie minimale.

Cette technique est à opposer avec la solution visant à organiser les nœuds sous forme de hiérarchie dans le modèle *via* l'utilisation de plusieurs groupes. Sémantiquement les nœuds ne font pas alors partie d'un même groupe de synchronisation et exploitent alors des nœuds passerelles pour faire transiter les informations. La solution multi-groupes est donc plus adaptée pour un usage de Kevoree sur les réseaux maillés P2P hybrides tandis que le *slicing* est plus adapté pour un usage de Kevoree sur un réseau P2P pur.

En conclusion la capacité de divergence est un outil efficace et commun aux protocoles et aux modèles d'architecture pour répondre à la forte volatilité des nœuds participants.

5.2 Validation expérimentale sur *cluster* de simulation

Une validation qualitative et quantitative a été effectuée sur cette implantation d'algorithme issue des travaux du monde de l'informatique distribuée, selon les indicateurs suivants :

- mesure du temps de propagation d'un nouveau modèle vers des nœuds membres d'un groupe ;
- capacité à résister à la perte de connexion entre nœuds ;
- capacité à détecter des modèles issus de modifications et donc de branches concurrentes et capacité à les réconcilier.

Pour chacun de ces indicateurs, un protocole expérimental a été mis en place pour simuler sur une grille de calcul les comportements d'un système vis-à-vis des reconfigurations envisagées dans le cas sapeur-pompier. Bien que la cible de production soit un environnement embarqué sur tablette de type Android, l'usage de la grille permet ici une simulation plus précise et à plus large échelle du comportement du réseau. Cependant le biais introduit ici ne permet de valider que le comportement de l'algorithme et non le temps de réponse absolu nécessaire pour le déploiement sur le terrain.

5.3 Protocole expérimental commun

Toutes les expériences suivantes partagent un protocole expérimental commun. Chacune utilise un ensemble de nœuds logiques Kevoree déployés sur un nœud plate-forme physique de la grille de calcul. Chaque nœud logique exploite donc sa propre machine virtuelle Java sur l'implémentation du nœud JavaSE de Kevoree . La grille expérimentale exploitée pour ces expériences est composée de nœuds de calcul hétérogènes en terme de type et de puissance. Les communications sont supportées par un réseau local à 100 MB/s.

5.3.1 Modèle de topologie initiale

Chaque expérience prend en entrée un modèle de démarrage qui décrit l'architecture de plate-forme de façon abstraite. Ce modèle décrit principalement la topologie des nœuds, leurs liens de communication disponibles ainsi que les instances du groupe sous test. Ce même modèle est exploité par les implantations de groupe de synchronisation pour débuter les communications de fragment à fragment. En construisant des modèles de topologie on peut alors influencer les communications des groupes et ainsi simuler des comportements vis-à-vis de topologies rencontrées dans le cas sapeur-pompier. On peut par exemple simuler des communications indirectes en supprimant un lien entre un nœud A et B, pour l'obliger à passer par un nœud tiers C. Ce modèle de topologie est donc clairement utile pour la simulation mais introduit un biais dans l'expérience, car on ne peut observer la construction de la topologie par le groupe lui-même. Ainsi dans un cas réel cette approche serait à remplacer par une approche similaire à T-Man de Jelasity *et al* [JB06a]. Les résultats de construction de topologie ne seront donc pas analysés ici en raison de ce biais.

5.3.2 Horloge de temps absolu pour la collecte des traces d'exécution

Afin de tracer la propagation des modèles et leur application dans le système, l'algorithme et l'implantation Java du groupe Gossip+VectorClock ont été décorés avec un *logger* en temps absolu. Celui-ci envoie des traces à chaque changement d'état interne, décrivant le groupe, le *VectorClock* courant et l'identification de la plate-forme origine de la propagation de modèle, ainsi que des métriques d'usage réseau. Afin d'exploiter les données temporelles de ces traces la synchronisation des traces effectue la réconciliation du temps d'émission avec une horloge de référence (en utilisant Java Greg Logger [4]). De façon plus précise, la synchronisation est fondée sur une architecture client/serveur, chaque client synchronisant alors de façon périodique sa latence observée avec le serveur référent. Ainsi chaque envoi peut réconcilier le temps absolu en prenant en compte cette latence observée plus la latence réseau. Puis les traces sont chaînées en observant les *VectorClock* récoltés et ainsi en ordonnant suivant l'ordre d'apparition des différentes versions prises par les modèles d'architecture. En résultat on dispose d'une liste de traces d'exécution retraçant les états temporisés des nœuds du cluster.

5.3.3 Mode de communication

Deux *patterns* d'échange sont principalement exploités pour construire l'algorithme. Le terme **période de pooling** est associé au temps passé entre deux synchronisations actives, initiées par un membre du groupe vers un autre. Dans cette phase de synchronisation un *VectorClock* ou un modèle est envoyé au membre origine.
La technique du **push/pull** est l'association d'une communication périodique et d'une communication par évènements. Ce mode permet l'envoi de notification à tous les groupes accessibles lorsqu'un nouveau modèle est prêt.

Trois expériences évaluent l'implantation de Kevoree et de son *GroupeType* Gossip+VectorClock. Ainsi on retrouve détaillées ci-dessous :

4. http://code.google.com/p/greg/

— l'expérience #1 visant à l'évaluation des temps de dissémination du modèles,
— l'expérience #2 évaluant la résistance aux fautes
— et l'expérience #3 évaluant la résistance à l'apparition de branches divergentes et la capacité de réconciliation de modèles.

5.4 Expérimentation 1 : Délai de propagation vis-à-vis de l'usage du réseau de communication

Cette première expérience cherche à mesurer de façon précise la capacité du *GroupType* à disséminer des modèles sur une topologie maillée. Cette mesure permet également de comprendre l'impact du délai de propagation vis-à-vis de l'usage de la bande passante réseau.

5.4.1 Protocole expérimental

Comme il est décrit dans la section du protocole expérimental commun, les mesures sont effectuées sur une grille dans un environnement équipé de sondes et contrôlé par un modèle de topologie inclus dans un modèle Kevoree . Après l'étape de démarrage sur ce modèle, un nœud est choisi périodiquement de façon aléatoire pour injecter un nouveau modèle dans le réseau. En pratique ceci se traduit par le calcul d'un nouveau modèle (par une machine extérieure pilotant le test) en simulant une migration d'une instance de composant d'un nœud à un autre (cas d'adaptation élastique). Ce nouveau modèle est réinjecté dans le nœud cible et le groupe instance est alors automatiquement notifié pour réaliser la propagation. Cette nouvelle configuration possède un tatouage qui permet la traçabilité de la source du modèle. La figure 5.1 montre le modèle de topologie utilisé pour cette expérience dédiée à la communication multi-saut (*multi-hop*).

Dans cette configuration statique 66 nœuds composent le réseau, et aucun n'est ajouté ni retiré durant l'expérience. Le modèle de topologie réseau influence directement et de manière significative les résultats, sa définition doit donc mettre en lumière les éléments à valider dans cette expérience dédiée au délai de propagation. Ainsi suivant les descriptions de Shudong *et al.* [JB06b] les dérivations de modèle *SmallWorld* en réduisant les liens directs permettent de simuler la communication multisaut et l'agrégation de nœuds, comme dans le cas du modèle utilisé en 5.1

Si le modèle de topologie est fixe l'expérience fait varier les paramètres suivants : délai de temps de *pooling* et changement du mode de communication *push/pull* ou uniquement *pull*. Deux lancements distincts permettent de tester les deux cas limites de l'algorithme, une première configuration sans les notifications avec un délai de synchronisation active de 1000 ms, et une deuxième avec activation et un délai de 15 s. Dans les deux cas, une reconfiguration est injectée toutes les 20 secondes, et 12 reconfigurations sont effectuées successivement.

5.4.2 Limites de validité expérimentale

Interne

Le surcoût introduit par les communications réseau lors de l'envoi de notifications est ici évalué dans son pire cas car réalisé par le biais d'un envoi multiple. Les sondes mesurent la quantité de données envoyées et reçues par les nœuds. Or dans le cas de réseau de type IP des mécanismes tels que le *multicast* permettent de réduire la recopie des données sur le réseau. Par extension ce type d'amélioration serait également disponible sur des technologies de réseau de type radio.

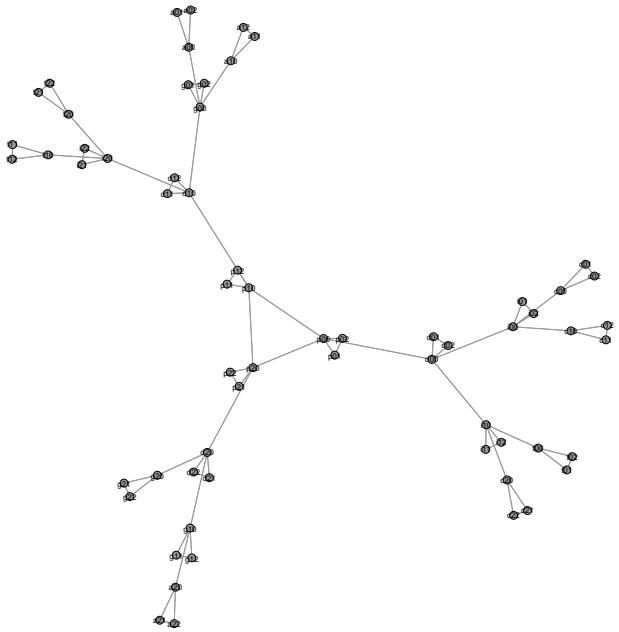


FIGURE 5.1 – Modèle de topologie, expérience #1

Externe

De par le caractère homogène du réseau interconnectant la grille de calcul, l'hétérogénéité de vitesse de transport n'est ici pas évaluée. Cependant l'hétérogénéité des nœuds de calcul introduit de fait une différence de temps de traitement qui permet d'observer et d'évaluer l'algorithme sur des temps de propagation variables.

5.4.3 Analyse des résultats expérimentaux

Les propagations mesurées par sauts sont présentées sous la forme d'un graphique indiquant leurs distributions percentiles 5.2.

Les valeurs représentées sont les valeurs brutes après réconciliation par l'horloge absolue divisées par le nombre de sauts entre la cible et l'origine du nouveau modèle émis (calculé suivant un algorithme de Bellman-Ford [CRKGLA89]). Le trafic réseau en volume de messages de protocole est représenté sur la figure 5.3 en KB par nœud et par reconfiguration. Ce volume n'inclut pas la charge utile. Les valeurs absolues de consommation réseau dépendent beaucoup de l'implantation du *GroupType* . Les résultats présentés ici sont inhérents à l'implantation Java et ils seraient bien évidemment différents pour l'implantation Android ou sur microcontrôleur. L'utilisation de notifications réduit significativement le délai de propagation moyen : celui-ci passe de 1510 ms/saut à 215 ms/saut. De plus, la représentation percentile montre clairement que l'écart-type propagation diminue avec l'emploi de notifications. En considérant uniquement le délai et non la charge réseau induite par l'inondation de messages, la capacité de passage à l'échelle de cette version est bien meilleure sur de grands *clusters*. Cependant en comparant cette version *push/pull* avec le *pull* simple, l'utilisation de notifications n'a pas un impact significatif sur le réseau. L'analyse plus fine des résultats a également permis d'associer des

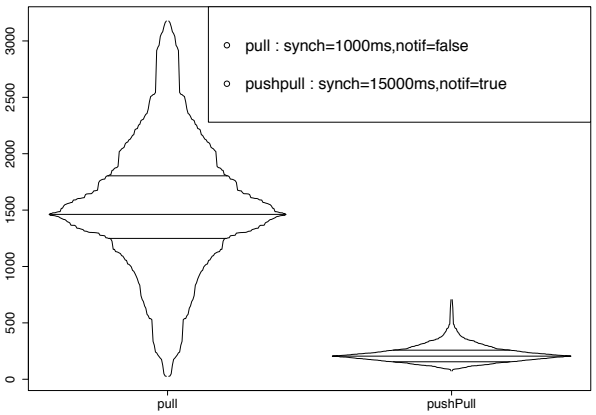


FIGURE 5.2 – Delai/hop(ms)

variations de cet impact suivant les types de cycle dans la topologie. En effet, la vitesse de propagation provoque la création de branches concurrentes au sein de sous-*clusters* puis leur diffusion. Ceci augmente le nombre de conflits à résoudre, même si l'algorithme réseau de dissémination est en revanche plus performant. Lorsque les notifications sont désactivées, les délais sont suffisamment longs pour éviter ces créations de branches concurrentes inutiles ; ceci supprime alors des transports réseaux non nécessaires et l'inondation par des modèles concurrents. Comme la charge utile de cet algorithme contient lui-même des informations de topologie (modèle Kevoree) il serait possible d'optimiser ces créations de branches en tenant compte de ces informations pour prévenir cette inondation. En résumé, en termes de délai de propagation le gain d'ajout de notification introduit un surcoût dans les communications réseau. Ce surcoût reste acceptable et peut être traité avec une extension de l'algorithme.

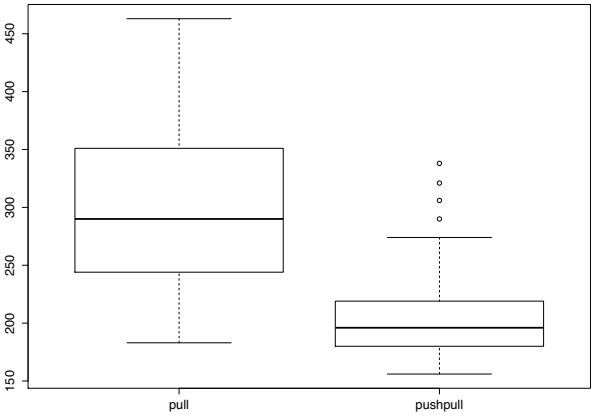


FIGURE 5.3 – Utilisation réseau/nœud(en kbytes)

5.5 Expérimentation 2 : Impact des erreurs de communication sur les délais de propagation

Un réseau mobile de terrain tel que celui exploité pour le système d'information tactique du cas sapeur-pompier est caractérisé par un un grand nombre de nœuds devenant fréquemment inaccessibles. L'algorithme à évaluer est justement défini pour résister à ces topologies à problèmes. Cette deuxième série de mesures évalue la capacité du groupe à disséminer un modèle dans un réseau possédant un fort taux de défaillance des liens entre nœuds.

5.5.1 Protocole expérimental

Ce protocole reprend les grandes lignes du précédent, le modèle de topologie est cependant remanié pour simuler un réseau maillé avec de nombreuses routes alternatives entre les nœuds (voir la figure 5.4).

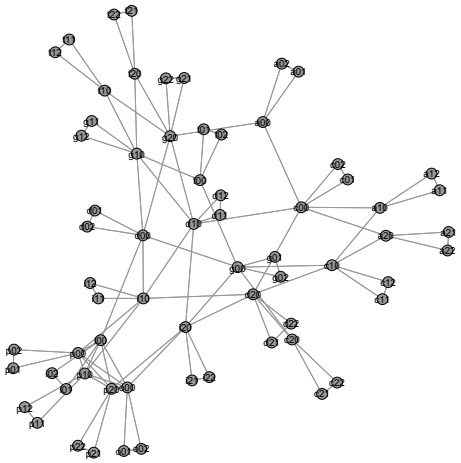


FIGURE 5.4 – Topologie pour expérience #2

De manière similaire à l'étape précédente un nouveau modèle représentant une migration de composants est injecté périodiquement. Durant chaque lancement de modèle, de nouvelles fautes de réseaux sont injectées entre les nœuds, selon une distribution de Poisson. Ainsi le taux d'erreur du réseau augmente à chaque modèle calculé, et le nombre de nœuds accessibles diminue.

Pour effectuer les simulations d'erreurs, des sondes sont injectées dans les machines virtuelles pour manipuler les cartes réseaux, ces sondes surveillent et synchronisent les événements de synchronisation envoyés à chaque exécution du protocole du groupe. À chaque modèle injecté une liste de nœuds théoriquement accessibles est calculée, on peut alors comparer cette liste avec les évènements réellement reçus par les autres nœuds pour calculer le temps de propagation moyen et vérifier la propagation globale.

5.5.2 Limites de validité expérimentale

Interne

De manière analogue à l'expérience précédente, l'homogénéité du réseau ne permet pas d'observer ici l'impact de temps de transport entre les nœuds de communication. Cependant le mode de propagation de type *gossip* employé ici n'exploite pas ni la valeur des débits, ni les choix de nœuds à synchroniser, ce qui limite l'impact de ce biais.

Externe

Les fautes sont injectées par le biais d'une désactivation du réseau, ce qui correspond à une panne totale de communication ou de noeud de calcul, l'évaluation d'une panne intermittente (avec un taux de pertes très élevé par exemple) n'est pas réalisée ici.

5.5.3 Analyse des résultats expérimentaux

La figure 5.5 illustre les résultats de l'expérience #2. L'histogramme qui y est représenté synthétise le taux de pannes réseau simulées à chaque lancement. La courbe rouge illustre le temps moyen de propagation (en millisecondes) à tous les noeuds atteignables. Au delà d'un taux de panne dans le réseau supérieur à 85%, le noeud origine de la nouvelle configuration est isolé du réseau et ceci termine alors l'expérience. En dessous de ce seuil, l'expérience montre que tous les noeuds atteignables reçoivent le nouveau modèle. Entre 0% et 60% le temps de propagation est variable dans une fourchette de 600 à 800 ms. Malgré ces variations dues essentiellement à la topologie et donc un nombre de sauts variables entre deux points, le taux de panne n'a pas d'impact décroissant sur le temps de dissémination.

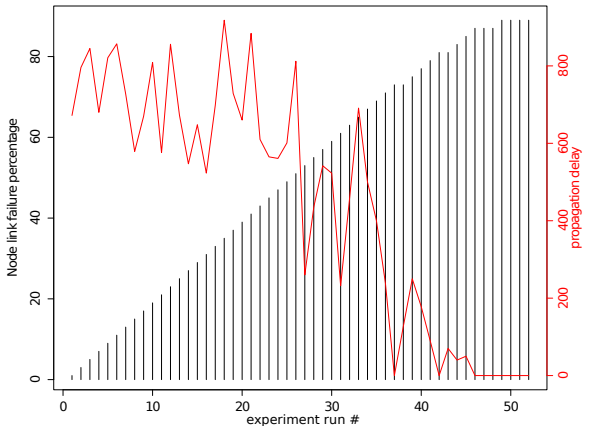


FIGURE 5.5 – Resultats expérience #2

5.6 Expérimentation 3 : Réconciliation de modèle et reconfigurations concurrentes

Cette dernière expérience évalue la capacité du groupe à détecter et réconcilier les modèles émis de façon concurrente par les nœuds. Ce type de problème arrive par exemple dans le cas

sapeur-pompier en partie à cause des communications sporadiques dans le réseau. Un nœud ou un groupe de nœuds peut alors être isolé pendant une période. En conséquence les reconfigurations distantes ne sont pas appliquées, de plus, de nouveaux modèles peuvent être calculés localement et diffusés dans le sous-réseau. Des reconfigurations concurrentes apparaissent alors au moment du rétablissement de la connexion . L'implantation du groupe se fonde alors sur la traçabilité des *VectorClock* pour détecter ces conflits directement sur les modèles émis. Notre expérience étudie cette réconciliation dans ce cas d'usage.

5.6.1 Protocole expérimental

Le protocole de l'étape précédente est simplifié pour exploiter uniquement une topologie de 12 nœuds, comme illustré par la figure suivante 5.6 :

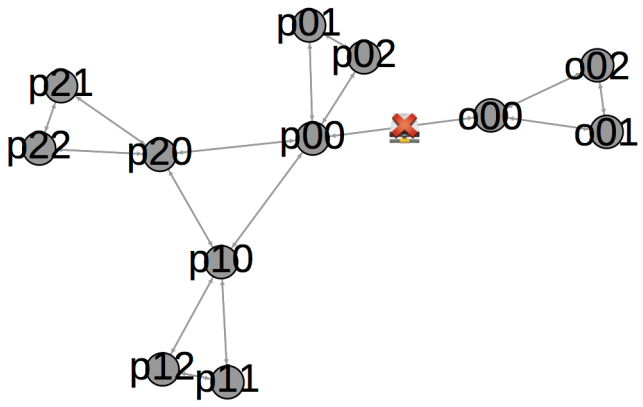


FIGURE 5.6 – Topologie pour l'expérience #3

Une configuration initiale *c1* est injectée dans un nœud *p00* juste après la phase de démarrage. Toutes les sondes mettent alors les liens réseaux en position valide et le modèle est alors propagé à l'ensemble des nœuds. Une faute est par la suite injectée entre les nœuds *p00* et *o00* et illustrée par une marque rouge sur le schéma. Les nœuds *o00*, *o01*, *o02* se retrouvent alors isolés. Un nouveau modèle est ensuite injecté au nœud *p00* (c2) et un modèle différent au nœud *o00* (c3). Un délai de 1000 ms sépare chaque -injection et l'algorithme est configuré avec des notifications et une période de *pooling* de 2000 ms.

5.6.2 Limites de validité expérimentale

Lors de la détection d'un conflit le nœud local doit alors faire une résolution qui correspond à l'assemblage des modifications stockées dans deux modèles différents, qu'il faut fusionner à l'aide de différentes stratégies, par exemple avec des règles de priorités. Le temps de calcul de cette résolution est considéré comme négligeable et non pris en compte ici, mais cependant suivant les stratégies de résolution ce temps peut s'avérer plus coûteux sur un très large réseau.

5.6.3 Analyse des résultats expérimentaux

La figure 5.7 illustre les résultats de cette troisième expérience. Ceux-ci sont directement dérivés de notre résultat d'arbres de traces.

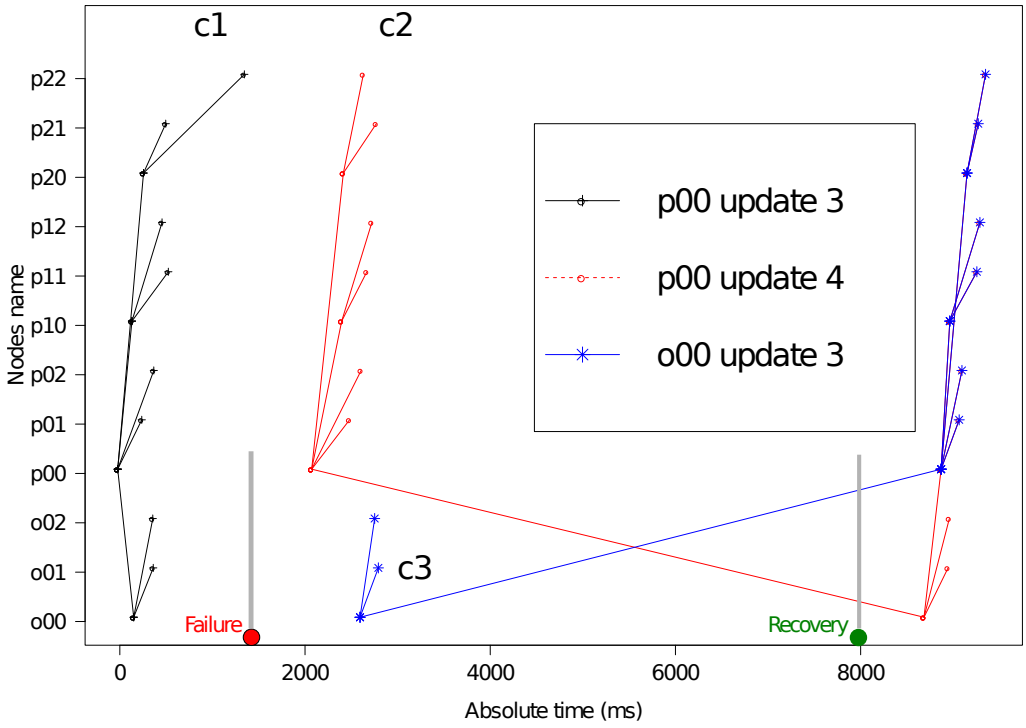


FIGURE 5.7 – Réconciliation de modèle émis en concurrence

Trois propagations de modèle sont représentées comme une succession de lignes qui représentent chacune la propagation d'un modèle d'un nœud à un autre. Le premier modèle envoyé sur le réseau sain et représenté en noir au temps 0 ms. Le troisième modèle injecté dans le nœud *o00* (au temps 2500ms) est représenté par la trace bleue tandis que le dernier injecté dans le nœud *p00* (time 2000ms) est illustré en rouge. Le premier modèle atteint directement tous les nœuds. Au temps 1500 ms la perte de connexion est alors simulée, le second modèle injecté en *p00* atteint tous les nœuds sauf ceux joignables uniquement *via o00*. De manière symétrique le second modèle qui est injecté dans le nœud *o00* n'est pas propagé aux nœuds après le nœud *p00*. Au temps 8000 ms, la simulation d'erreur de communication commencée au temps 1500 ms est annulée. Après un délai de synchronisation de 380 ms on observe la réconciliation du modèle et sa propagation dans sa forme fusionnée à tous les nœuds *via* la trace représentée en violet.

5.7 Conclusion sur l'usage des groupes pour la convergence

A défaut de pouvoir proposer une évaluation exhaustive des implantations possibles des *GroupType* Kevoree dans les domaines des algorithmes de dissémination distribués, cette section a présenté un cas limite mettant en lumière les capacités de divergences et convergences de Kevoree . L'aspect exhaustif et l'état de l'art sur l'écosystème Kevoree sera abordé plus en détails dans la section suivante permettant ainsi d'évaluer la maturité de l'évaluation de l'abstraction vis-vis d'autres implantations. Cette série d'expériences permet cependant de mettre en lumière l'adéquation des résultats obtenus avec les besoins du cas sapeur-pompier. En effet que ce

soit sur les délais réseau ou la capacité de détection et correction des modèles concurrents l'implantation résiste correctement aux paliers demandés. De manière plus générale ceci valide également l'hypothèse que le surcoût introduit par un style de programmation opportuniste laissant diverger le système est envisageable, et peut largement s'appuyer sur les résultats du distribué pour être amélioré et exploiter des heuristiques adaptées à chaque cas d'usage. Enfin ceci met là encore en lumière le besoin d'exprimer la diversité des algorithmes de convergence qui par définition existe de part ces variantes.

Chapitre 6

Models@runtime pour le cloud computing

Ce chapitre présente un ensemble d'expérimentations pour utiliser le modèle de configuration de Kevoree pour piloter des adaptations multi-niveaux dans le domaine du cloud computing. Ces expérimentations, menées dans le cadre de la thèse d'Erwan Daubert, ont pour but de i) valider l'extensibilité de l'approche de models@runtime pour différents fournisseurs de clouds afin d'utiliser le modèle de configuration dans le cadre de fédérations de clouds, ii) valider la capacité à capturer la complexité liée à la configuration d'une application déployée dans le cloud, et iii) confirmer la pertinence de l'utilisation d'un modèle de configuration commun aux différents niveaux d'un cloud (Infrastructure, plateforme, application).

Sommaire

6.1 Expérimentation 4 : Est-ce extensible et générique ? **81**
6.1.1 Protocole expérimental 81
6.1.2 Implémentation du cas d'étude 82
6.1.3 Évaluation 85
6.2 Expérimentation 5 : Est-ce utilisable pour le pilotage infrastructure de Clouds ? **86**
6.2.1 Protocole expérimental 87
6.2.2 Implémentation du cas d'étude 88
6.2.3 Évaluation 95
6.3 Expérimentation 6 : Est-ce utilisable pour de l'adaptation multi-niveaux ? . **97**
6.3.1 Protocole expérimental 97
6.3.2 Mise en œuvre du cas d'étude 97
6.3.3 Définition du serveur web distribué 98
6.3.4 Évaluation 99
6.4 Synthèse **101**

Le concept de Cloud Computing ou "informatique nuagique" consiste à fournir les ressources informatiques sous forme de services pour lesquels l'utilisateur paie pour ce qu'il utilise. Ce

concept est apparu dans les années 60 notamment avec McCarthy [GA99] ou encore Kleinrock [Kle05] sous le nom d'informatique utilitaire. C'est ensuite vers la fin des années 90 que ce concept a réellement pris de l'importance avec tout d'abord le Grid Computing [FK03]. Ce terme est une métaphore exprimant la similarité avec le réseau électrique dans lequel l'électricité est produite dans de grandes centrales puis disséminée au travers d'un réseau jusqu'aux utilisateurs finaux. Ici les grandes centrales sont les DataCenters, le réseau est le plus souvent celui d'Internet et l'électricité correspond aux ressources informatiques. Le terme *Cloud Computing* n'est véritablement apparu qu'au cours des années 2006-2008 [Vou08] avec l'apparition d'Amazon EC2 [Ama] ou encore la collaboration d'IBM et Google [IBMa, IBMb] ainsi que l'annonce d'IBM concernant 'Blue Cloud' [IBMc]. Par la suite de nombreuses solutions open source ont aussi vu le jour avec par exemple OpenShift [Opea] de RedHat, ou encore OpenStack [Opeb] de RackSpace et en collaboration avec la NASA.

La problématique de configuration dans le cloud est souvent un enjeu important que l'on soit un fournisseur d'infrastructure, un fournisseur de plate-forme ou un fournisseur d'applications multi-tenants ou multi entités déployées dans le cloud permettant à la même application de servir plusieurs organisations clientes. Pour illustrer cette complexité de configuration, prenons l'exemple d'OpenStack discuté en Section 2.1.

L'extrait d'architecture présenté en figure 2.1, présentée en Section 2.1, montre la complexité potentielle de telles architectures. Si l'on modélise ensuite la plate-forme d'exécution et les applications afin de pouvoir envisager des adaptations transverses et coordonnées, la complexité globale et l'hétérogénéité de tel système se perçoit vite. Une des expériences a donc été de confronter notre langage de configuration Kevoree et l'ensemble de l'approche de modélisation à l'exécution pour maîtriser la complexité de tels systèmes.

Afin de valider l'abstraction proposée et aborder ainsi la question de recherche RQ5, nous montrons dans ce chapitre un compte rendu d'expérience visant à démontrer trois points.

— Nous cherchons à démontrer que le modèle proposé est suffisamment générique et extensible pour pouvoir concevoir facilement de nouvelles implantations d'infrastructure ou de plate-forme de Cloud.
— Nous questionnons ensuite les problématiques de passage à l'échelle en vérifiant que les temps de traitement liés à la boucle autonomique [LMD13] construite à partir d'une approche de modèles à l'exécution restent raisonnables.
— nous montrons que notre abstraction permet bien de gérer l'adaptation comme une problématique transverse et qu'elle offre la possibilité de concevoir des adaptations multi-niveaux, c'est-à-dire des adaptations ayant des impacts à la fois sur l'infrastructure, la plate-forme d'exécution et le niveau applicatif.

Afin de démontrer ces différents points, ce chapitre relate trois expériences. La première, en section 6.1, présente l'implantation d'une infrastructure de Cloud hybride à base de virtualisation d'espaces utilisateurs pour des systèmes Unix et Linux et de virtualisation matérielle pour les systèmes Microsoft. Cette expérimentation présente aussi l'implantation d'un gestionnaire d'infrastructure dont l'objectif est d'utiliser les systèmes de virtualisation proposés pour héberger au mieux les plates-formes et de manière à équilibrer la charge sur l'ensemble des ressources concrètes offertes par l'infrastructure. Dans cette expérimentation, nous évaluons la quantité de lignes de code nécessaire pour définir un nouveau type de nœuds d'infrastructure ainsi qu'un gestionnaire d'infrastructure dans le but de montrer la facilité de concevoir des nouveaux types de nœuds et des nouveaux types de gestionnaire.

La seconde expérimentation, en section 6.2, présente l'implantation d'une plate-forme capable de répartir les composants d'une application sur une infrastructure en prenant en compte les contraintes concernant les ressources nécessaires au bon fonctionnement des composants. Dans cette expérimentation, nous évaluons l'impact de Kevoree au niveau du temps d'exécution

des reconfigurations ainsi que de l'empreinte mémoire pour le stockage du modèle architectural représentant l'ensemble du Cloud (Infrastructure, Plate-forme et Applications).

Enfin la troisième expérimentation, en section 6.3, présente l'implantation d'une plate-forme pour la gestion de serveur web distribué. Dans cette expérimentation, nous évaluons comment l'usage de Kevoree permet de partager une vision globale du système permettant à l'infrastructure et la plate-forme de tirer parti des informations de l'ensemble du système. Nous montrons aussi comment il est possible de construire des systèmes d'adaptations coordonnées en utilisant le modèle architectural comme support de coordination.

Certaines expérimentations ont été menées sur Grid5000, cependant la reproductibilité des expérimentations n'étant pas assurée principalement concernant le temps d'exécution, nous avons mené l'ensemble des expérimentations présentées dans ce chapitre sur 10 machines identiques à base de processeur Intel R, Atom™ D425 1.8GHz et avec 8 Go de mémoire vive (voir figure 6.1).

FIGURE 6.1 – Machines physiques ayant servi de Cloud

6.1 Expérimentation 4 : Est-ce extensible et générique ?

L'objectif de cette section est de montrer que l'utilisation de notre abstraction permet de faciliter la définition de nouveaux types de nœud ainsi que de nouveaux gestionnaires d'infrastructure et de plate-forme.

6.1.1 Protocole expérimental

Dans cette évaluation, nous allons présenter l'implantation de plusieurs types de nœuds pour le niveau infrastructure ainsi qu'un gestionnaire d'infrastructure capable de distribuer les nœuds de plate-forme sur les nœuds d'infrastructure. Afin d'évaluer l'extensibilité de notre *framework*, nous avons compté le nombre de lignes de code nécessaires à l'implantation d'un nouveau type de nœud (sans compter le code généré) et comparons ces résultats aux nombres de lignes de code fournies par Kevoree ainsi que par KevoreeKloud et qui permettent au nouveau type de nœuds de réutiliser directement une implantation des différentes fonctionnalités qui doivent être implantées. Les chiffres concernant le nombre de lignes de code ont été obtenus par l'intermédiaire de l'outil [1]

1. http://cloc.sourceforge.net/

6.1.2 Implémentation du cas d'étude

Il existe plusieurs types de Cloud, nous allons ici nous intéresser aux Clouds hybrides qui permettent aux entreprises de combiner leurs propres ressources avec celles fournies par un Cloud public. L'usage du Cloud public permet d'augmenter ponctuellement le nombre de ressources utilisées lorsque les ressources locales ne sont plus suffisantes.

Dans cette section, nous allons présenter l'implantation de quelques types de nœuds ainsi que la définition du gestionnaire d'infrastructure qui permet de gérer un Cloud hybride dans lequel l'ensemble des machines virtuelles Windows sera hébergé sur une solution de virtualisation matérielle, ici sur Amazon EC2 et les machines virtuelles Linux et Unix seront hébergées sur une solution de virtualisation d'espaces utilisateurs, ici en utilisant les Jails de FreeBSD ou les conteneurs Linux (LXC ou Docker.io).

6.1.2.1 Infrastructure d'espace utilisateur

Nous avons choisi d'implémenter une solution d'infrastructure utilisant les Jails [2] de FreeBSD [3]. Les Jails ont été intégrées comme solution avancée de *chroot* [4]. Elles permettent d'exécuter les services sensibles dans des environnements distincts de celui du système d'exploitation. Contrairement au *chroot* qui ne peut isoler que le système de fichiers en définissant une racine spécifique pour les processus s'exécutant dans le *chroot*, les *Jails* permettent un plus haut niveau d'isolation en offrant la capacité d'isoler non seulement le système de fichiers, mais aussi les interfaces réseaux. Les *Jails* sont aussi capables de limiter la consommation des ressources que ce soit en terme d'espace disque, de quantité de mémoire vive et de temps CPU ou de nombre de cœur de CPU. Tout comme Amazon EC2 qui fournit des *AMIs* correspondant à des configurations de base pour des machines virtuelles, l'environnement Jails permet de définir ce que l'on appelle des *flavors* qui permettent de définir des patrons (*templates*) de *Jails*. Ces patrons permettent de définir le système exécuté dans les *Jails*, c'est-à-dire quels sont les services au démarrage, quels sont les applications disponibles, quels sont les utilisateurs enregistrés.

Le type *JailNode* (voir listing 6.1) permet donc de définir une *Jail* pour chaque nœud de plate-forme hébergé par l'instance du nœud d'infrastructure. En plus des primitives définies dans *IaaSNode*, un *JailNode* est aussi capable de sauvegarder la configuration d'une *Jail* et permet aussi de recharger une configuration préalablement sauvegardée.

Listing 6.1 – Définition du JailNode

```
@PrimitiveCommands(
  values = {"SAVE_NODE", "RELOAD_NODE"})
@DictionaryType({
    @DictionaryAttribute(name = "default_mode"),
    @DictionaryAttribute(name = "default_flavor")
})
@NodeType
public class JailNode extends IaaSNode {
```

Dans le cadre de cette expérimentation, nous avons aussi défini un type de nœud de plate-forme pouvant définir des caractéristiques aux Jails (voir listing 6.2). Ces caractéristiques sont prises en compte par le nœud d'infrastructure au moment de la création des *Jails*. Ainsi, il est possible de spécifier une *flavor* particulière ainsi qu'une archive qui correspond à une sauvegarde d'une *Jail* précédente.

2. http://www.freebsd.org/doc/en/books/handbook/jails.html
3. http://www.freebsd.org/
4. http://en.wikipedia.org/wiki/Chroot

Listing 6.2 – Définition du PJailNode

```
@DictionaryType({
    @DictionaryAttribute(name = "flavor", optional = true),
    @DictionaryAttribute(name = "archive", optional = true),
})
@NodeType
public class PJailNode extends MiniCloudNode implements PaaSNode {
```

Pour ces deux types de nœuds, les propriétés héritées au travers de *IaaSNode* et *PaaSNode* permettent aussi de définir les caractéristiques concernant la quantité de mémoire vive ainsi que le processeur.

En plus du type de ces types de nœud, nous avons aussi défini un composant de gestion d'infrastructure (voir listing 6.3). Son rôle est de définir le lien d'hébergement entre les nœuds de plate-forme ajoutés dans le modèle par le gestionnaire de plate-forme et les nœuds d'infrastructure présents dans le système. Nous avons défini une implantation basique dans laquelle les nœuds de plate-forme sont alloués sur les nœuds d'infrastructure en fonction du nombre de nœuds déjà hébergés. De cette manière, chaque nœud d'infrastructure héberge le même nombre de nœuds de plate-forme même si ceux-ci ne possèdent pas les mêmes limitations.

Bien entendu cette implantation est très naïve, mais la thèse d'Erwan Daubert [Dau13] n'avait pas pour objectif de travailler sur des algorithmes de placement intelligent.

6.1.2.2 Infrastructure de container systèmes

LXC, contraction de l'anglais *Linux Containers* est un système de virtualisation, utilisant l'isolation comme méthode de cloisonnement au niveau du système d'exploitation. Il est utilisé pour faire fonctionner des environnements Linux isolés les uns des autres dans des conteneurs partageant le même noyau et une plus ou moins grande partie du système hôte. Le conteneur apporte une virtualisation de l'environnement d'exécution (Processeur, Mémoire vive, réseau, système de fichier...) et non pas de la machine. Pour cette raison, on parle de « conteneur » et non de machine virtuelle.

LXC repose sur les fonctionnalités des *Cgroups* du noyau Linux disponibles depuis la version 2.6.24 du noyau. Il repose également sur d'autres fonctionnalités de cloisonnement comme le cloisonnement des espaces de nommage du noyau, permettant d'éviter à un système de connaître les ressources utilisées par le système hôte ou un autre conteneur.

LXC propose deux mécanismes pour démarrer soit un système complet (lxc-execute) soit uniquement une application (*lxc-start*). Au démarrage de ces commandes, de nombreuses options de configuration doivent être fournies pour limiter plus ou moins l'accès au système hôte et limiter l'accès aux ressources.

Docker est un projet open source qui automatise le déploiement d'applications dans des conteneurs logiciels. Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur virtuel, qui pourra être exécuté sur n'importe quel Linux. Docker est construit par dessus LXC et en particulier *lxc-execute* et fournit une API de haut niveau. Contrairement aux machines virtuelles traditionnelles, un conteneur Docker n'inclut pas de système d'exploitation, à la place il s'appuie sur les fonctionnalités du système d'exploitation fourni par l'infrastructure sous-jacente. Même si docker fournit une abstraction par rapport à LXC, chaque conteneur reste très configurable et chaque application inclue dans un conteneur embarque sa propre logique de configuration.

Comme pour les Jails, nous avons construit trois implantations de types de nœrespectivement pour LXC, LightLXC (*lxc-execute*) et Docker en prenant en offrant une abstraction

Listing 6.3 – Définition du gestionnaire de l'infrastructure

```
@ComponentType
class IaaSManager extends AbstractComponentType implements ModelListener {
  def modelUpdated() {
    val iaasModel = getModelService.getLastModel
    val kengine = getKevScriptEngineFactory.createKevScriptEngine
    // count current child for each parent nodes
    val parents = KloudModelHelper.countChilds(iaasModel)
    var potentialParents = List[String]()
    var doSomething = false
    var usedIps = Array[String]()
    // filter nodes that are not IaaSNode and are not hosted by an IaaSNode
    iaasModel.getNodes.filter(n => KloudModelHelper.isPaaSNode(iaasModel, n.getName)
      && iaasModel.getNodes.forall(parent => !parent.getHosts.contains(n))).foreach {
      // select a host for each user node
      node => {
        if (potentialParents.isEmpty) {
          // get a list of nodes that have less child nodes than the others
          potentialParents = lookAtPotentialParents(parents)
        }
        val parentName = selectParent(potentialParents)
        kengine.addVariable("nodeName", node.getName)
        kengine.addVariable("parentName", parentName)
        kengine append "addChild {nodeName}@{parentName}"
        potentialParents = potentialParents.filterNot(p => p == parentName)
        doSomething = true
      }
    }
    if (doSomething) {
      getModelService.unregisterModelListener(this)
      try {
        // apply the KevScript on the current model to build a new one and send it to the Kevoree Core
        updateIaaSConfiguration(kengine)
      } catch {
        case ignored : SubmissionException=>
      } finally {
        getModelService.registerModelListener(this)
      }
    }
  }
  def selectParent(potentialParents : List[String]) : String = {
    // randomly select one of the potential parent nodes
    val index = (java.lang.Math.random() * potentialParents.size).asInstanceOf[Int]
    val parentName = potentialParents(index)
  }
}
```

commune pour la gestion des ressources et des options de configuration propre à chaque type de conteneur. Ces mises en œuvre sont disponibles ici [5].

6.1.2.3 Proxy pour infrastructure EC2

L'utilisation des *Jails* de FreeBSD ou de LXC permet d'avoir de la virtualisation d'espace utilisateur. Cependant, il n'est pas possible d'héberger des nœuds dans lesquels s'exécuteraient des applications Windows. C'est pourquoi nous avons choisi de réutiliser une solution existante permettant de créer des nœuds de plate-forme avec un système d'exploitation Windows. Pour cela, nous avons choisi de définir un proxy utilisant l'API EC2 permettant de se connecter au Cloud d'Amazon mais aussi à n'importe quel Cloud compatible avec cette API (par exemple CloudStack, OpenNebula, Nimbus).

Avec ce proxy (voir listing 6.4), nous offrons la possibilité de créer de nouvelles instances, d'en arrêter ou d'en redémarrer ou encore d'en supprimer. Ce type de nœud possède en plus

5. https://github.com/kevoree/kevoree-library/tree/master/cloud

des caractéristiques héritées du *ProxyIaaSNode*, des propriétés lui permettant de définir les caractéristiques par défaut des instances de machines virtuelles qu'il peut déployer.

Listing 6.4 – Définition du EC2Node

```
@DictionaryType({
  @DictionaryAttribute(name = "DEFAULT_INSTANCE_IMAGE")
  @DictionaryAttribute(name = "DEFAULT_INSTANCE_TYPE", optional = true)
})
@NodeType
public class EC2Node extends ProxyIaaSNode {
```

Nous avons aussi étendu le gestionnaire de l'infrastructure pour qu'il n'envoie les nœuds de plateforme sur le Cloud EC2 que si ces nœuds de plate-forme nécessitent un Windows en tant que système d'exploitation (voir listing 6.5).

Listing 6.5 – Algorithme pour le déploiement des système Windows

```
node =>
  if (node.getDictionary().get("OS").toLowerCase().contains("microsoft") ||
      node.getDictionary().get("OS").toLowerCase().contains("windows")) {
    val iaaSNodeWithVirtualizationTypeNodes = getModelService.getLastModel.getNodes.filter(n =>
        KloudModelHelper.isASubType(n.getTypeDefinition, "ProxyIaaSNode"))
    if (iaasNodeWithVirtualizationTypeNodes.size > 0) {
      val potentialParents = List[String]()
      iaaSNodeWithVirtualizationTypeNodes.foreach{n => potentialParents = potentialParents ++
          List[String](n.getName)}
      val parentName = selectParent(potentialParents)
      kengine.addVariable ("nodeName", node.getName)
      kengine.addVariable ("parentNodeName", parentName)
      kengine.append("addChild {nodeName}@{parentNodeName}")
    } else {
      logger.error("Unable to host {} because there is no IaaS able to host such kind of node.", node.getName)
    }
  }
```

6.1.3 Évaluation

Le tableau de la figure 6.2 récapitule les chiffres que nous avons obtenus sur les différentes implantations de type (*JailNode*, *LXCNode*, *LightLXCNode*, *DockerNode*, *EC2Node*, *IaaSManager*) et sur les APIs et implantations réutilisées (Kevoree API, JavaSeNode, Kevoree Core). Nous constatons que la définition du type *JailNode* correspond à 529 lignes de code qui contient l'implantation nécessaire pour créer et supprimer une *Jail* mais aussi du code capable de définir et modifier les contraintes posées sur une *Jail* (CPU, RAM). Concernant le proxy vers le Cloud d'Amazon, nous avons 322 lignes de code correspondant à l'ajout et la suppression de machines virtuelles ainsi que l'observation des machines virtuelles déjà créées. Le gestionnaire d'infrastructure (*IaaSManager*) nécessite quant à lui 316 lignes de code pour effectuer le placement des nœuds de plate-forme. L'API que nous avons défini dans Kevoree pour la gestion de conteneur contient 967 lignes de codes permettant de définir un ensemble d'abstractions de type ainsi qu'une implémentation spécifique de la phase de planification incluant les primitives "ADD_NODE" et "REMOVE_NODE". Le type *JavaSENode* est quant à lui composé de 2547 lignes de code représentant l'algorithme de planification de base ainsi que l'implantation de l'ensemble des primitives de reconfiguration concernant les composants, les canaux de communication et les groupes. Enfin le Kevoree Core qui offre la gestion du modèle à l'exécution et

la gestion du processus d'adaptation ainsi que le langage de script permettant de définir des reconfigurations est composé de 21768 lignes de code.

Le tableau de la figure 6.3 montre les ratios de code par rapport aux codes réutilisés (*Kevoree API*, *JavaSeNode*, *Kevoree Core*). Il montre aussi les ratios de code par rapport au framework de Kevoree lui-même.

Projet	**Nombre de lignes de code**
JailNode	529
LXCNode	780
LightLXCNode	351
DockerNode	1287
EC2Node	322
IaaSManager	316
KevoreePlatformAPI	967
JavaSENode	2547
Kevoree Core	21768

FIGURE 6.2 – Nombre de lignes de code non générées selon le projet

Projet	**Ratio par rapport aux codes réutilisés (%)**	**Ratio par rapport au framework lié à la gestion de conteneurs(%)**
JailNode	2,17	54,70
LightLXCNode	1,30	36,29
LXCNode	3,20	80,66
DockerNode	5,20	133,09
EC2Node	1,32	33,29
IaaSManager	1,29	32,67
KevoreePlatformAPI	3,97	100

FIGURE 6.3 – Ratio de code selon le projet

Nous pouvons constater que grâce à la réutilisation et l'héritage des types, la définition de nouveaux types nécessite peu de code par rapport à l'implantation de base avec *JavaSENode* ainsi que par rapport au gestionnaire de modèle (*Kevoree Core*). De même, le gestionnaire d'infrastructure nécessite lui aussi peu de code principalement grâce au gestionnaire de modèle qui offre de nombreuses fonctionnalités pour manipuler le modèle, mais aussi grâce au langage KevScript qui permet d'exprimer simplement des modifications de la configuration.

De plus, le gestionnaire d'infrastructure n'est en rien directement dépendant des types de nœuds mais bien des types abstraits proposés par l'API de Kevoree pour la gestion de conteneurs. De cette manière, ce gestionnaire peut tout à fait être réutilisé sur une autre infrastructure. Par exemple, il est envisageable de remplacer le *proxy EC2* par un autre *proxy* pour Microsoft Azure par exemple et de remplacer le type *JailNode* par un type *LXCNode*.

6.2 Expérimentation 5 : Est-ce utilisable pour le pilotage infrastructure de Clouds ?

L'objectif de cette expérimentation est de montrer que l'utilisation de l'abstraction proposée basée sur les techniques de modèle à l'exécution n'a pas d'impact négatif sur le temps de reconfiguration nécessaire dans le cadre d'un Cloud. Ce temps de reconfiguration correspond au temps nécessaire pour le déploiement de nouvelles machines virtuelles intégrées dans une plate-forme existante. C'est ce temps de déploiement qui définit l'instant à partir duquel la plate-forme peut utiliser ces machines virtuelles pour déployer de nouveaux composants.

6.2.1 Protocole expérimental

Dans cette évaluation, nous allons présenter l'impact de notre abstraction sur le temps de déploiement de nouveaux nœuds de plate-forme ainsi que l'impact sur la quantité de mémoire vive utilisée pour stocker le modèle de configuration. Pour cela, nous avons déployé des applications de grandes tailles sur notre infrastructure (logicielle et matérielle) de Cloud. Le déploiement de ces applications implique le déploiement d'un certain nombre de nœuds de type plate-forme.

Le déploiement d'un nœud de type plate-forme correspond à la mise en place d'un système virtualisé sur l'un des nœuds de l'infrastructure. Dans notre cas d'étude, ce déploiement consiste à créer un espace utilisateur sur l'une des machines FreeBSD. Ce déploiement peut aussi consister à déployer une machine virtuelle sur le Cloud EC2 ou un nouveau conteneur Docker.

Afin de montrer l'impact sur le déploiement des *Jails*, nous avons mesuré le temps nécessaire à la mise en place de la configuration d'un cas d'étude réel qui utilise 50 nœuds de plate-forme déployés sur 10 nœuds d'infrastructure. Pour mesurer le temps de déploiement, nous avons intégré, aux gestionnaires de plate-forme et d'infrastructure, un mécanisme de trace (*logger*) en temps absolu qui émet des événements vers un serveur qui effectue une réconciliation du temps par rapport à une horloge de référence (voir Java Greg Logger [6]). Ce serveur permet à chaque client de synchroniser périodiquement le temps de latence entre l'horloge de référence et celle du client permettant ensuite au serveur de prendre en compte cette latence et la latence réseau pour calculer un temps absolu.

Les traces publiées sont émises à partir des endroits ci-dessous :

— **trace 1** : réception du modèle par le gestionnaire de plate-forme
— **trace 2** : soumission d'un nouveau modèle par le gestionnaire de plate-forme après définition des nœuds de plate-forme nécessaire
— **trace 3** : réception du modèle par le gestionnaire d'infrastructure
— **trace 4** : soumission d'un nouveau modèle par le gestionnaire d'infrastructure après définition de l'hébergement des nœuds de plate-forme non alloués
— fin de la reconfiguration sur un nœud

Ces différentes traces permettent de calculer le temps nécessaire pour chaque opération que ce soit la génération d'un modèle par le gestionnaire de la plate-forme, la génération d'un modèle par l'infrastructure et la mise en place de la reconfiguration.

Nous comparons ensuite l'ensemble des temps obtenus pour estimer le surcoût (*overhead*) introduit par Kevoree et qui correspond au temps de génération des modèles.

Outre les résultats sur une application réelle, nous avons simulé des modèles de tailles variées afin de montrer l'impact de la taille du modèle sur l'utilisation mémoire de notre système de Cloud. Pour cela, nous avons aussi généré un modèle avec seulement cinq nœuds qui correspond à un sous-ensemble du modèle de l'application réelle. Nous avons ensuite généré quatre autres modèles respectivement de 500, 5000, 50000 et 100 000 nœuds. Ces modèles étant trop importants par rapport à la taille de notre infrastructure, la reconfiguration n'a pu être finalisée, mais nous avons tout de même obtenu les temps de traitement du modèle (il manque le temps de démarrage des *Jails* qui ne correspond pas au surcoût introduit par Kevoree).

Nous avons aussi évalué l'impact de notre abstraction sur la quantité de mémoire vive nécessaire à son bon fonctionnement. Nous avons mesuré ici la taille mémoire utilisée pour représenter la configuration globale du Cloud en mémoire afin d'évaluer l'impact du modèle Kevoree sur l'utilisation mémoire. Pour cela, nous avons mesuré la taille en mémoire des modèles précédemment utilisés dans l'évaluation de l'impact sur le temps de déploiement. Nous avons, en plus de ces modèles, synthétisé un modèle pouvant représenter la configuration du Cloud

6. http://code.google.com/p/greg/

d'Amazon selon les données de Guy Rosen [Guy] pour qui Amazon EC2 comptait autour de 50 000 machines virtuelles en septembre 2009 ainsi que selon les données de Huan Liu [Hua] pour qui Amazon EC2 comptait autour de 7000 servers en Mars 2012.

Pour finir, nous avons aussi évalué l'extensibilité et la généricité de notre solution en regardant le nombre de lignes de code nécessaires pour concevoir cette nouvelle plate-forme de Cloud.

Finalement, comme dernier critère d'évaluation, nous avons de nouveau mesuré le nombre de lignes de code nécessaire pour la définition de ces nouveaux types de nœud et de gestionnaire d'infrastructure.

6.2.2 Implémentation du cas d'étude

Pour disposer d'applications nécessitant de manière réelle un grand nombre de ressources potentiellement isolées, nous avons choisi de traiter un cas d'étude particulier qu'est le test logiciel dans le cadre de l'intégration continue.

L'intégration continue [FF06] est un principe de génie logiciel visant à suivre l'évolution du développement d'un logiciel. En pratique, cela consiste principalement à automatiser la compilation de l'application et les phases de tests. Ainsi, l'ensemble des développeurs intègre leurs modifications sur un serveur de source (Git, SVN, Mercurial) et pour chaque commit, le serveur d'intégration effectue une vérification de ces modifications afin d'assurer que l'application compile et que les tests soient tous valides. Si quelque chose échoue, les acteurs autour du projet sont capables de savoir quelle est la modification ayant entraîné cet échec et il est ainsi plus facile de pouvoir la corriger.

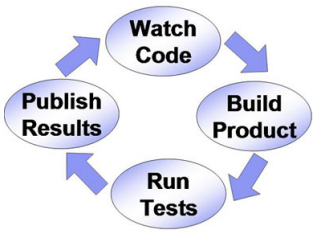


FIGURE 6.4 – Processus d'intégration continue

De nombreux projets possèdent plusieurs centaines et même milliers de tests permettant d'assurer la stabilité du logiciel. Par exemple, Rothermel *et al.* affirme, dans leur article [RUCH01], que l'exécution des tests unitaires de certains projets de leurs collaborateurs industriels nécessitait jusqu'à 7 semaines.

Ce genre de durée ne peut correspondre à l'idée mise en avant dans l'intégration continue qui est de pouvoir détecter rapidement les modifications entraînant des régressions et des erreurs afin de pouvoir les corriger tout aussi rapidement. Plusieurs travaux ont été menés pour tenter de diminuer ce temps. Il existe par exemple des approches essayant de sélectionner un sous-ensemble des suites de tests à exécuter en fonction des modifications apportées [GHK+98, WHLA97]. Un autre type d'approche est de définir des priorités sur l'exécution des tests en exécutant les tests les plus importants d'abord [WHLA97, KP02]. Ces techniques ont pour objectifs de détecter le plus tôt possible les régressions ou erreurs dues aux modifications. Enfin il y a les techniques cherchant à distribuer l'exécution des tests [Kap01, DCBM06].

Mais outre la problématique de la durée d'exécution des tests, il y a aussi la problématique des contraintes d'exécution de ces tests. En effet, si un logiciel doit être multi plate-formes, il

doit pouvoir s'exécuter sur plusieurs systèmes différents, que ce soit en termes de système d'exploitation, de processeur, de mémoire vive, etc. De même, certains tests ont besoin d'isolation afin de permettre la bonne exécution du reste du processus. Un exemple simple est l'utilisation de la Kinect au travers de l'API freenect [7] codée en C. Cette API propose aussi un wrapper Java qui possède trois tests unitaires. Cependant, l'exécution de chacun des tests peut terminer de manière anormale le processus de la machine Java à cause d'un *bug* de l'API qui produit une erreur de segmentation lors de la tentative de connexion à la Kinect et que celle-ci n'est pas connectée à la machine. Si la machine Java part en erreur, alors il n'est pas possible d'obtenir le moindre résultat concernant les tests puisque le processus d'exécution s'est terminé de manière anormale.

L'utilisation du Cloud Computing dans le cadre du tests d'applications complexes peut-être une réponse à cette problématique. En effet, il est possible de réserver autant de machines nécessaires pour pouvoir exécuter les tests sur l'ensemble des configurations requises. Les serveurs d'intégration continue actuels fournissent ce genre de capacité. C'est le cas par exemple de Jenkins et de plusieurs de ses *plugins* qui permettent de réserver des instances spécifiques sur des infrastructures de Cloud telles que Amazon EC2 ou encore DeltaCloud puis d'exécuter la compilation et le test des projets sur ces instances. Mais cette solution d'interaction avec une infrastructure reste une solution ad hoc au plugin qui la propose et brise l'encapsulation offerte par une plate-forme puisque c'est l'application qui directement demande à l'infrastructure de lui fournir de nouvelles ressources (nœuds de plate-forme) pour s'exécuter.

De plus, la distribution des tests n'est pas prise en compte puisque la répartition se fait par projet et non pas par suite de tests et encore moins par tests. Ainsi, pour un projet devant être testé sur différentes plates-formes, l'ensemble des tests va être exécuté pour chacune des configurations alors que seulement un sous-ensemble doit prendre en compte cet aspect (par exemple, les tests d'une interface GTK sur un système ayant la bibliothèque GTK). Là encore, une première solution consiste à modulariser le projet afin de pouvoir définir des processus d'intégration continue sur chacun des modules. C'est déjà l'approche qui est utilisée dans un grand nombre de projets. Cela permet de limiter le temps d'exécution de la compilation et du test de chaque module et permet aussi de répartir l'exécution de chaque module afin de les exécuter en parallèle et sur des plates-formes spécifiques si nécessaire. Cependant, cette modularisation est effectuée deux fois. Tout d'abord lors de la définition du projet et ensuite lors de la définition de son intégration continue. À chaque modification de la modularisation du projet, il est nécessaire de modifier les processus d'intégration continue. Enfin, si un module doit être testé dans différents environnements, il est nécessaire de définir ces environnements et de définir les processus d'intégration continue correspondants.

La définition des processus d'intégration continue selon les environnements pourrait pourtant être évitée ou au moins simplifier puisque la majorité des informations sont connues lors de la modularisation du projet. La définition d'autant de processus d'intégration continue que de plates-formes à tester peut, de plus, être automatisée en spécifiant les différents critères pour chacun des modules ou même pour chacune des suites de tests voir pour chacun des tests unitaires. La spécification de critères peut aussi être intéressante dans le cadre de l'exécution isolée des tests. C'est le cas dans le cadre du test d'application Java ayant du code natif qui s'exécute comme l'usage du *wrapper* Java de *Libfreenect*. En effet, dans ce wrapper, il y a trois tests qui vont entraîner un crash de la machine virtuelle dans lesquels ils s'exécutent si la kinect n'est pas connectée à la machine empêchant ainsi l'exécution des autres tests existants. Avoir la possibilité d'exécuter chacune des suites de tests de manière indépendante ou même chaque test unitaires de manière isolée peut assurer que l'ensemble des tests soit effectué même si certains tests sont en erreur. Plus simplement cela permet d'obtenir un résultat sur l'exécution des tests

7. http://openkinect.org/wiki/Main_Page

montrant ceux qui ont échoué.

Nous avons donc développé une extension (*plugin*) pour le serveur d'intégration continue *Jenkins*[8] permettant à partir d'un projet de synthétiser une architecture d'application Kevoree (un modèle de configuration) regroupant l'ensemble des tests à exécuter. Cette architecture d'application est soumise à une plate-forme spécifique capable de déployer l'exécution de chaque test sur des nœuds de la plate-forme en fonction de leurs contraintes d'exécution. La plate-forme est capable de demander des ressources à l'infrastructure sous-jacente sans pour autant nécessiter une connaissance particulière de celle-ci. Pour cela, nous avons défini un composant de gestion de la plate-forme capable de définir de nouveaux nœuds de plate-forme n'étant pas hébergés par l'infrastructure. Cette définition de nouveaux nœuds se traduit par la proposition d'un nouveau modèle pour le système.

L'infrastructure et son gestionnaire sont donc notifiés de la mise en place de ce nouveau modèle. Le gestionnaire (voir *listing* 6.3) que nous avons défini suivant la philosophie du protocole *map-reduce* [DG08] est lui capable de détecter que des nœuds de type plate-forme ne sont pas hébergés par l'infrastructure. Il peut donc décider de les placer sur les ressources de l'infrastructure.

6.2.2.1 Plate-forme de déploiement de tests unitaires

Il existe plusieurs *frameworks* pour la définition de tests. Nous pouvons citer par exemple *JUnit*[9] ou *TestNG*[10] pour les tests unitaires en Java.

JUnit permet de définir des tests unitaires avec la possibilité de définir une gestion de cycle de vie des tests et des paramètres d'exécution de ces tests par l'intermédiaire de plusieurs annotations spécifiques (*@Before*, *@BeforeClass*, *@After*, *@AfterClass*). TestNG propose quant à lui un peu plus de fonctionnalités dont entre autres la capacité à paralléliser l'exécution des tests sur une même machine en utilisant des threads différents. Il est indiqué aussi sur le site du projet que TestNG est capable de distribuer l'exécution des tests sur un ensemble de machines sur lesquelles s'exécuterait un esclave TestNG. Cependant, cette fonctionnalité est limitée et ne semble pas avoir été maintenue ou en tout cas n'est pas mise en avant sur les versions actuelles. De même que TestNG, GridUnit [DCBM06] propose aussi de distribuer l'exécution des tests et plus précisément de distribuer les tests sur une grille de calcul.

Bien que TestNG fournisse quelques fonctionnalités intéressantes pour effectuer de l'exécution parallèle voire distribuée des tests, cela n'est pas suffisant. En effet, outre le fait de vouloir paralléliser et distribuer les tests à exécuter, nous souhaitons aussi pouvoir spécifier les caractéristiques des plates-formes d'exécution. De plus, en aucun cas TestNG ne permet d'utiliser des infrastructures de Cloud pour déployer les plates-formes et nécessite une configuration statique de l'ensemble des machines permettant de distribuer les tests (voir le blog de l'auteur[11] pour plus de détails).

Du fait que ni JUnit, ni TestNG ne permettent de spécifier l'ensemble des informations nécessaires pour pouvoir exécuter les tests unitaires de manière distribuée, parallèle et isolée, nous avons défini un framework supplémentaire pour la définition de tests unitaires et plus particulièrement pour la définition des caractéristiques d'exécution de ces tests.

Pour cela, nous proposons un DSL interne défini à l'aide d'un ensemble d'annotations Java. Ces annotations permettent de spécifier les propriétés nécessaires de la plate-forme d'exécution (voir listing 6.6).

8. http://jenkins-ci.org
9. http://www.junit.org/
10. http://testng.org/
11. http://beust.com/weblog2/archives/000362.html(accessibleenjanvier2013)

Listing 6.6 – Annotations pour la configuration de la plate-forme d'exécution

```
@OS {values = {"Windows XP", "Ubuntu 12.04", "FreeBSD 9"}}
@RAM {values = {"512MB", "1GB"}}
@CPU_CORE {values = {"2"}}
@CPU_FREQUENCY {values = {"3GHz"}}
```

Il est ainsi possible de définir le type de système d'exploitation, le nombre de cœurs du processeur, la fréquence du processeur et la quantité de mémoire disponible. Chacune de ces annotations peut prendre plusieurs valeurs afin de spécifier l'ensemble des possibilités pour chaque caractéristique. Ces caractéristiques sont ensuite composées pour définir l'ensemble des systèmes sur lesquels le test ou la suite de tests doit être exécuté. Il est aussi possible de spécifier des configurations spécifiques plutôt qu'un ensemble de propriétés qui seront composées (voir listing 6.7).

Listing 6.7 – Annotation pour la définition de configuration spécifique

```
@Configuration ({
  @OS {values = {"FreeBSD 9"}},
  @RAM {values = {"512MB"}},
  @CPU_CORE {values = {"2"}},
  @CPU_FREQUENCY {values = {"3GHz"}}
})
```

En plus de ces annotations spécifiant les contraintes de la plate-forme d'exécution, nous proposons aussi un ensemble d'annotations permettant de définir des caractéristiques non plus sur l'environnement d'exécution, mais sur l'exécution elle-même (voir listing 6.8). Ces annotations permettent de spécifier si le test ou la suite de tests doit s'exécuter de manière isolée afin d'assurer que si l'exécution échoue, elle n'aura pas d'impact sur l'exécution des autres tests comme cela peut être le cas avec les tests sur la *Kinect*. Il est aussi possible de définir une notion de dépendance entre différents tests. Cette dépendance permet d'expliciter le fait que si le test A a échoué alors cela ne sert à rien d'exécuter le test B puisque celui-ci utilise la fonctionnalité testée par A. En plus de l'isolation des tests et de leur dépendance, il est possible de définir que des tests peuvent être exécutés en parallèle d'autres tests. Cette information signifie que hormis pour les tests, dont le test A dépend, il est possible d'exécuter le test A avec d'autres tests sur la même plate-forme. Cette propriété est intéressante notamment dans le cadre de tests unitaires ne cherchant pas à tester les performances mais simplement la fonctionnalité en elle-même. Enfin dans le cadre du test de performance, nous proposons de spécifier la durée maximale autorisée pour l'exécution d'une fonctionnalité.

Avec ce *framework* de test, nous fournissons une suite d'outils intégrée au sein d'un plugin maven capable de synthétiser, à partir d'un projet, le modèle de configuration pour Kevoree regroupant l'ensemble des composants correspondant à une suite de tests ou un test. Le modèle de configuration définit au travers des canaux de communication les relations entre les composants. De cette manière, un composant ne peut exécuter le test associé que lorsqu'il reçoit un message explicite. Ce message est envoyé par les composants hébergeant les tests qui sont en dépendance du test hébergé. Une fois que le composant a reçu les messages de tous les composants dont il dépend, il peut exécuter le test qu'il héberge. En plus des composants représentant les différents tests et les canaux de communication représentant les dépendances de test, l'architecture logicielle contient un composant spécifique chargé de la gestion des tests. Cette gestion se caractérise par le démarrage des tests et l'agrégation des résultats. Ainsi, le composant de gestion

Listing 6.8 – Annotations pour la définition de configuration spécifique

```
@Configuration ({
  @VMARGS {values = {"-Xms512m", "-Xmx1024m", "-XX:PermSize=256m", "-XX:MaxPermSize=512m"}},
  @ISOLATED
  @DEPENDS_ON ({
    @Test {className = org.kevoree.test.TestSuite1.class, testName = "test1"},
    @Test {className = org.kevoree.test.TestSuite2.class, testName = "test2"}
  }),
  @PARALLEL
  @TIMEOUT {values = {"10000"}}
})
```

est connecté à tous les composants n'ayant pas de dépendance afin de pouvoir les déclencher comme le font les composants entre eux. Les composants de tests émettent aussi les résultats des tests qu'ils envoient au gestionnaire sous la forme de données XML conformes à une grammaire communément utilisée par les outils autour de Junit. Ce format est ainsi proposé à l'origine par la tâche ant de JUnit [12] et qui est aussi proposé par le plugin maven Surefire [13] développé par Apache. C'est aussi un format souvent utilisé par les plugins de jenkins pour manipuler les résultats de tests.

L'architecture logicielle définit aussi les contraintes sur les plates-formes d'exécution. Pour cela, les composants sont hébergés par des nœuds ayant les caractéristiques associées aux tests et plusieurs tests peuvent être hébergés sur le même nœud. En plus des caractéristiques de plate-forme, l'annotation *ISOLATED* permet aussi de définir une contrainte sur l'exécution puisqu'elle spécifie que le test doit être exécuté dans un processus spécifique. Pour cela, nous avons défini un type de nœud que nous avons appelé *MiniCloudNode* qui permet de créer de nouvelles machines virtuelles Java sur le même système (voir listing 6.9 pour la définition du type).

Listing 6.9 – Définition du type MiniCloud

```
@DictionaryType({
    @DictionaryAttribute(name = "VMARGS", optional = true)
})
@NodeType
public class MiniCloudNode extends HostNode {
}
```

Une fois l'architecture définie, la suite d'outil développé dans le cadre de cette expérimentation envoie le modèle à la plate-forme d'hébergement. Cette plate-forme d'hébergement définit le déploiement des composants sur la plate-forme en fonction du modèle de configuration fourni, des nœuds disponibles et des nœuds qu'il peut créer (voir listing 6.10).

Pour tenir compte des relations entre les composants, et sachant que ce gestionnaire est un gestionnaire spécialisé dans la distribution des tests, il construit un graphe de relation entre les composants qui correspond au graphe de dépendances entre les tests ou les suites de tests. Il place ensuite les composants de façon à limiter le nombre de tests qui s'exécute en même temps sur un nœud de plate-forme. Pour cela, il choisit de positionner un composant et au moins l'un de ces successeurs sur le même nœud. De cette façon, si le composant B dépend du composant A alors le composant A devra finir de s'exécuter avant que B puisse commencer. La figure 6.5 représente un exemple de répartition de composants sur les nœuds et incluant les relations

12. http://ant.apache.org/manual/Tasks/junit.html
13. http://maven.apache.org/plugins/maven-surefire-plugin/

Listing 6.10 – Définition du gestionnaire de la plate-forme

```
@ComponentType
class PaaSManager extends AbstractComponentType implements ModelListener {
  def process(newPaaSModel : ContainerRoot) {
    val graph = buildDependencyGraph(newPaaSModel)
    val index = new DirectedNeighborIndex(graph)
    val firstStep = graph.vertexSet().filter(v => index.predecessorsOf(v).size() == 0).toList
    var number = 0
    var alreadySeenList = List[ComponentInstance]()
    var previousStep = firstStep
    var currentModel = getModelService.getLastModel
    while (number < graph.vertexSet().size()) {
      var newStep = List[ComponentInstance]()
      previousStep.foreach {
        p =>
          val nodeName = findNodeForComponent(p)
          var first = true
          val tmp = index.successorListOf(p).filter(v => index.predecessorsOf(v).forall(pred =>
                alreadySeenList.contains(pred))).toList
          newStep = newStep ++ tmp
          tmp.foreach {
            component =>
              val kengine: KevScriptEngine = getKevScriptEngineFactory.createKevScriptEngine(currentModel)
              val componentName = component.getName
              if (first) {
                first = false
                // add the first sucessor on the same node than its current predecessor
                addComponentOnNode(componentName, nodeName, currentModel, kengine)
              } else {
                // try to find an existing node to host the component
                if (!chooseEquivalentNode(currentModel, model, nodeName, kengine)) {
                  // create a new node to host the component
                  val tmpNodeName = findNodeForComponent(p)
                  val newNodeName = buildNode(tmpNodeName, model, kengine)
                  addComponentOnNode(componentName, newNodeName, currentModel, kengine)
                }
              }
              // apply changes (on a temporary model) to take into account into the next iteration
              currentModel = kengine.interpret()
          }
      }
      alreadySeenList = alreadySeenList ++ newStep
      number = number + newStep.size
      previousStep = newStep
    }
    try {
      getModelService.atomicCompareAndSwapModel(uuidModel, currentModel)
    } catch {
      case e : Throwable =>
    }
  }
}
```

d'ordre (correspondant aux dépendances) entre ces composants. Sur cette figure, nous pouvons voir que le composant *Camel Core* ne possède aucune dépendance et que quasiment l'ensemble des autres composants dépend de lui. Ainsi, l'exécution débutera par le *Camel Core*. Une fois que celui-ci sera terminé, il notifiera les composants qui lui sont connectés afin qu'ils débutent leur exécution et ainsi de suite.

6.2.2.2 Résultat sur un projet concret : Apache Camel

Nous avons effectué notre expérimentation sur le projet Apache Camel [14]. Apache Camel est un *framework* qui implante l'ensemble des EIPs pour Enterprise Integration Patterns [HW04] en

14. http://camel.apache.org/

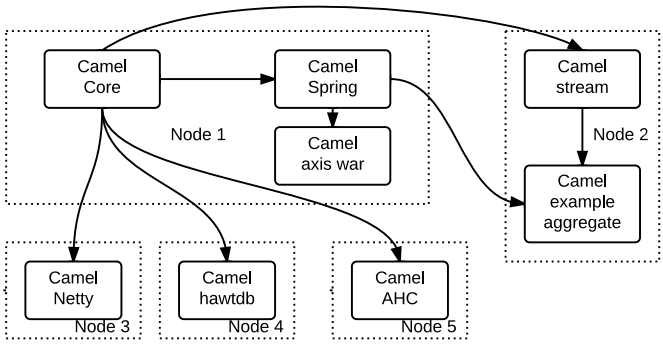


FIGURE 6.5 – Graphe de dépendances et allocation des composants sur les nœuds

anglais. Ces patrons décrivent l'ensemble des opérations couramment utilisées dans le cadre de l'intégration d'applications. Apache Camel propose un DSL permettant d'utiliser ces patterns afin de simplifier la construction des routines de communication permettant l'intégration de deux logiciels. Pour notre expérimentation, ce projet a l'avantage de compter 8084 tests dans la version 2.8 pour un temps d'exécution des tests d'environ une heure et quarante minutes.

Dans notre expérimentation, nous avons choisi de découper l'exécution des tests au niveau des suites de tests qui sont au nombre de 3845 pour 175 modules. Nous n'avons pas spécifié de dépendance sur l'isolation de l'exécution de ces suites de tests, ainsi plusieurs suites de tests peuvent être exécutées dans la même machine virtuelle Java. De même, nous n'avons pas spécifié de dépendances vis-à-vis d'un système d'exploitation afin que l'ensemble des nœuds de plate-forme soit créé en tant que *Jails*. Outre les caractéristiques que nous n'avons pas spécifiées, l'ensemble des suites de tests partage aussi trois caractéristiques communes que sont les propriétés pour la machine virtuelle Java (-Xmx1024m -XX :MaxPermSize=512m). Ces spécificités proviennent de la documentation du projet Apache Camel qui nous informe des contraintes nécessaires sur la machine virtuelle Java pour exécuter l'ensemble des tests. Ces contraintes fournissent par la même occasion une contrainte sur la quantité mémoire que doivent posséder les nœuds de plate-forme (plus de 1024Mb). La troisième contrainte identique correspond au fait que l'ensemble des tests peut être parallélisé. Enfin, la dépendance entre les suites de tests a été définie en fonction des dépendances entre les différents modules du projet Apache Camel. Cette contrainte va permettre de définir les interactions entre les différents composants.

Comme notre Cloud d'expérimentation est limité en terme de ressource, nous avons choisi de limiter le nombre de nœuds de plate-forme à 50 soit 5 par nœud d'infrastructure. Dans notre cas d'utilisation, le gestionnaire de placement est capable de créer 50 nœuds le gestionnaire de placement retourne obligatoirement un de ces nœuds pour héberger le composant. Pour cela, cette méthode utilise les informations disponibles dans le modèle lui permettant de savoir que notre infrastructure se compose de 10 machines ayant chacune 8Gb de mémoire. Bien que cette contrainte n'ait pas d'intérêt tel quel dans une utilisation sur une infrastructure telle qu'Amazon, elle pourrait être remplacée par une limitation en fonction du prix que l'utilisateur serait prêt à payer.

Au final, nous avons donc 3845 composants à exécuter sur 50 Jails qui s'exécutent en 35 minutes (soit un speedup de 2.8) sachant que le test le plus long prend 20 minutes et qu'il nous faut environ 7 minutes pour démarrer les 50 Jails. Cette accélération qui peut paraître limitée s'explique par les dépendances entre les composants. Ces dépendances ne permettent pas l'exécution des tests de manière parallèle.

Les résultats sur ce cas concret nous montrent que la distribution et la parallélisation de l'exécution des tests unitaires sont possibles. En effet, bien que ce projet ne nécessite qu'une heure et quarante minutes pour exécuter les tests, l'accélération de 2.8 nous permet d'envisager un gain (en terme de temps) non négligeable sur des projets nécessitant plusieurs jours.

6.2.3 Évaluation

6.2.3.1 Impact sur le déploiement

Le tableau de la figure 6.6 montre les délais entre la réception de modèle par le gestionnaire de plate-forme et les différentes étapes de la mise en place de ce nouveau modèle.

	5	50	500	5000	50000	100000
trace 1	0	0	0	0	0	0
trace 2	27	176	2188	32185	337638	684650
trace 3	193	361	2571	34920	345263	693205
trace 4	222	543	4766	67115	682914	1377870
trace 6	223537	425306				

FIGURE 6.6 – Temps des réceptions des traces

La figure 6.7 représente les différentes données du tableau précédent sous forme de courbes.

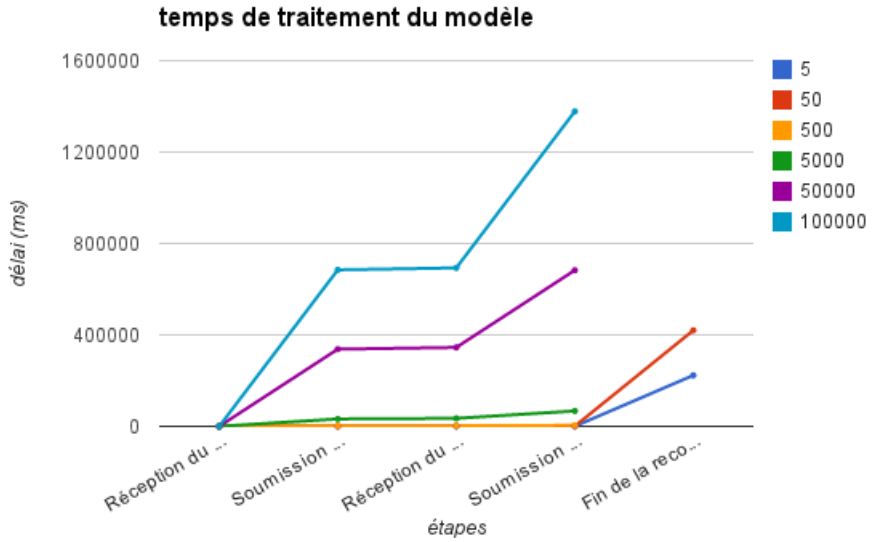


FIGURE 6.7 – Temps de traitement du modèle

Ces données nous montrent que dans le cadre du projet *Camel* (la courbe pour 50 nœuds et finissant autour des 400000 ms), le traitement du modèle utilisateur pour définir les nœuds de plate-forme utilisés pour le déploiement des tests est sensiblement le même que pour le positionnement de ces nœuds sur l'infrastructure. Nous pouvons noter aussi que le temps nécessaire pour le passage entre la trace 2 et la trace 3 reste faible en comparaison des autres délais. Cela s'explique par le fait que même si les nœuds de plate-forme sont définis, ils ne sont en aucun

cas hébergés et donc le système n'a pas besoin d'appliquer une mise à jour concrète. Ici la soumission d'un nouveau modèle par la plate-forme tire parti du fonctionnement du *Kevoree Core* et notamment de la gestion des *ModelListeners* pour notifier l'infrastructure du changement. C'est cette notification qui permet de déclencher l'exécution du gestionnaire de placement.

Si l'on compare les temps de traitement du modèle avec la mise en place des nœuds de plate-forme (ici les jails), nous pouvons voir que le temps de traitement est faible en comparaison avec la mise en place des jails. Cependant, si l'on regarde les temps de traitement pour les modèles de l'ordre de 50 000 et 100 000 nœuds, le temps nécessaire pour définir l'hébergement des nœuds de plate-forme sur l'infrastructure devient beaucoup plus important. Cela est principalement dû au fait de devoir, pour chaque nœud de plate-forme, sélectionner une IP qui n'est pas encore utilisée dans le réseau de l'infrastructure. Si nous prenons les mêmes modèles, mais que seulement 5 nœuds ont besoin d'un hébergement (les autres sont déjà en place) alors ce temps est fortement réduit.

6.2.3.2 Impact sur l'utilisation mémoire

Le tableau de la figure 6.8 récapitule les chiffres correspondants à la taille mémoire des différents modèles que nous avons évalués.

Modèle	**taille (Mega octets)**
50 nœuds et 4000 éléments	3,72
500 nœuds et 40000 éléments	35,88
5000 nœuds et 100000 éléments	91,97
5000 nœuds et 200000 éléments	160,47
5000 nœuds et 300000 éléments	269,58
5000 nœuds et 400000 éléments	357,47
100000 nœuds et 400000 éléments	424,44
507000 nœuds et 400000 éléments	721,21

FIGURE 6.8 – Espace mémoire pour le stockage d'un modèle

Nous pouvons voir que l'utilisation d'un modèle global est quelque chose d'utilisable pour des systèmes de cette taille mais les ressources, ici la mémoire, utilisées sont tout de même importantes.

Pour autant, les chiffres qui ont été mesurés correspondent à des modèles complets ne tirant pas parti des capacités de projection de modèles. En effet, l'ensemble des informations conservées dans les modèles peut tout à fait être limité. Par exemple, même si l'infrastructure peut se servir de la définition des interactions entre les composants pour positionner les nœuds hébergeant ces composants, il n'est pas nécessaire de connaître les caractéristiques de l'implémentation de ces composants. Il était donc possible de faire de la projection de modèles permettant de limiter les informations de types et de paramétrage contenu dans le modèle.

De même, dans le cadre de cette expérimentation, l'implémentation de notre représentation modèle n'est en aucun cas optimisée pour des systèmes de très grande taille. Il serait intéressant de confronter ces modèles face aux nouvelles propositions faites autour de *Kevoree Modelling Framework* [HFN+14].

6.2.3.3 Complexité de l'implémentation de nouveaux types et gestionnaire

Le tableau de la figure 6.9 présente le nombre de lignes de code pour un certain nombre de types.

Le gestionnaire de plate-forme dans ce cas d'étude contient 309 lignes de code (sans compter le *plugin maven* pour la génération du modèle de l'application cliente). Là encore, nous pouvons

Projet	nombre de lignes de code
PJavaSENode	12
PMiniCloudNode	12
MiniCloudNode	150
PaaSManager	309

FIGURE 6.9 – Nombre de lignes de code selon le projet

constater que la définition d'un gestionnaire de plate-forme nécessite peu de lignes de code grâce à la conception modulaire et extensible de Kevoree. Concernant l'implantation des types de nœuds de plate-forme, nous utilisons le type *PJavaSENode* ainsi que *PMiniCloudNode* qui hérite respectivement de *JavaSENode* et *MiniCloudNode*. Ces deux types de nœuds ne redéfinissent rien, que ce soit pour la la planification ou pour l'exécution des primitives car ils possèdent les mêmes caractéristiques que les types parents avec en plus un héritage du type PaaSNode qui permet de définir des caractéristiques spécifiques pour les nœuds de plate-forme. Ainsi, ces types contiennent chacun 12 lignes de code et le type MiniCloud contient 150 lignes de code.

6.3 Expérimentation 6 : Est-ce utilisable pour de l'adaptation multi-niveaux ?

L'objectif de cette section est de montrer que l'utilisation du modèle de configuration proposé permet bien de gérer l'adaptation comme une problématique transverse en permettant la définition de politiques d'adaptation multi-niveaux. Afin de supporter l'adaptation multi-niveaux, il faut premièrement que chaque moteur de raisonnement puisse avoir accès aux informations concernant l'ensemble des niveaux. Deuxièmement il est nécessaire que l'ensemble des moteurs de raisonnement soit capable de collaborer pour définir les adaptations les plus efficaces.

6.3.1 Protocole expérimental

Dans cette évaluation, nous allons présenter comment le fait d'avoir accès un support commun pour la configuration des différents niveaux permet d'envisager des adaptations plus performantes.

Tout d'abord, nous allons montrer que l'accès aux informations de ces différents niveaux permet d'envisager des adaptations plus efficaces. Pour cela, nous allons définir un gestionnaire d'infrastructure capable de tenir compte des connexions entre composants pour placer à proximité les nœuds de plate-forme hébergeant ces composants.

Nous montrons ensuite que le fait d'avoir un support commun pour la définition des reconfigurations permet la collaboration des moteurs de raisonnement. Dans le cas d'étude, nous montrons que la collaboration entre les moteurs de raisonnement chargés de l'infrastructure et de la plate-forme permet de limiter les migrations de machines virtuelles en les remplaçant par des migrations de composants ou par la définition de composant de cache permettant de limiter les requêtes vers le composant qui utilise beaucoup de bande passante.

6.3.2 Mise en œuvre du cas d'étude

L'élasticité et la scalabilité sont en général définies comme deux caractéristiques importantes d'un Cloud. En effet, l'un des intérêts de migrer ses applications sur un Cloud est de pouvoir assurer aux utilisateurs une qualité de service suffisante. Pour cela, la plate-forme d'hébergement est le plus souvent capable de dynamiquement assurer cette qualité de service par l'intermédiaire

de duplication des composants de l'application et de l'utilisation de techniques de *load-balancing*. Ainsi, un serveur web distribué sera défini comme un point d'accès que l'on peut appeler le serveur et un ensemble de composants capables de générer les pages du ou des sites hébergés.

L'implantation de ce cas d'étude se découpe en deux parties. Nous allons tout d'abord présenter comment sont définis un serveur web et les pages web hébergées. Ces pages web sont des pages statiques. Nous présenterons ensuite l'implantation du gestionnaire d'élasticité permettant de gérer la duplication ou la migration de composants avec en plus la possibilité de créer de nouveaux nœuds de plate-forme en fonction des besoins.

6.3.3 Définition du serveur web distribué

Pour implémenter ce cas d'étude, nous avons défini deux types de composants.

Le premier type définit l'interface d'une page web (voir listing 6.11) qui permet de traiter les requêtes qu'elle reçoit et de retourner des réponses à ces requêtes. Cette page web définit aussi l'URL relative dont elle a la charge.

Listing 6.11 – Définition d'une page web

```
@Provides({
    @ProvidedPort(name = "request", type = PortType.MESSAGE)
})
@Requires({
    @RequiredPort(name = "content", type = PortType.MESSAGE),
    @RequiredPort(name = "forward", type = PortType.MESSAGE, optional = true)
})
@DictionaryType({
    @DictionaryAttribute(name = "urlpattern", optional = true, defaultValue = "/")
})
@ComponentFragment
public abstract class AbstractPage extends AbstractComponentType {
```

Le second type définit l'interface d'un serveur web qui écoute sur un port, qui envoie les requêtes client qu'il reçoit à ceux capables de les traiter, puis lorsqu'il reçoit les réponses correspondantes, les envoie aux clients (voir listing 6.12).

Listing 6.12 – Définition d'un serveur web

```
@DictionaryType({
    @DictionaryAttribute(name = "port" , defaultValue = "8080"),
    @DictionaryAttribute(name = "timeout" , defaultValue = "5000", optional = true)
})
@Requires({
    @RequiredPort(name = "handler", type = PortType.MESSAGE)
})
@Provides({
    @ProvidedPort(name = "response", type = PortType.MESSAGE)
})
@ComponentFragment
public abstract class AbstractWebServer extends AbstractComponentType {
```

La définition d'un site web consiste donc à associer un serveur avec un ensemble plus ou moins grand de pages web (voir figure 6.10 pour un exemple).

C'est sur ce modèle que nous avons construit l'expérimentation du projet Diversify, [15].

15. cloud.diversify-project.eu

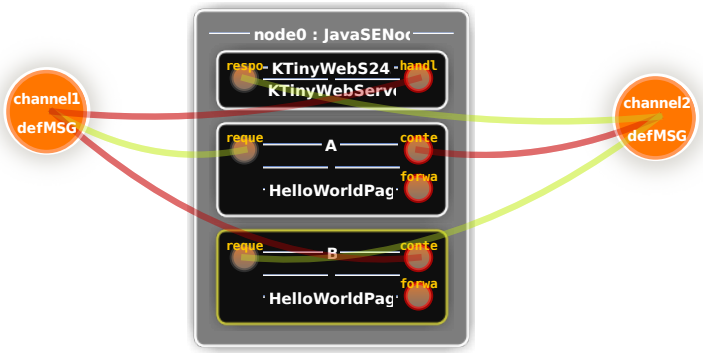


FIGURE 6.10 – Configuration de base du serveur web

Définition de la plate-forme Les types de nœuds de plate-forme restent les mêmes que dans le cas d'étude précédent. C'est le gestionnaire qui change pour prendre en compte la notion d'élasticité de l'application. Ce gestionnaire a pour objectif d'assurer la qualité de service nécessaire au serveur web. Pour cela, un nœud capable d'observer son exécution et notamment son usage de mémoire et de temps processeur est utilisé.

Ainsi, lorsque le gestionnaire détecte que la consommation de ressources d'un nœud devient critique, il est en charge de déplacer, ou dupliquer les composants de page web afin de décharger le nœud (voir listing 6.13). En plus de cela, il est nécessaire qu'il mette en place des canaux de communication spécifiques capables de distribuer les requêtes entre les composants. Pour cela, nous avons défini un *channel* de type *RoundRobin* qui permet de sélectionner, de manière aléatoire, le composant à qui la requête est soumise. La limitation du nombre de nœuds de plate-forme est actuellement fixée en tant que paramètre du gestionnaire mais ce paramètre pourrait être calculé dynamiquement en fonction du coût des nœuds de plate-forme et du coût que l'utilisateur de la plate-forme accepte de payer.

L'implémentation de ce gestionnaire reste simple puisque nous ne surveillons que la charge du processeur et il serait nécessaire d'avoir une implantation plus complète pour être capable de gérer efficacement cette notion d'élasticité. Par exemple, il serait sans doute nécessaire d'observer le temps nécessaire entre l'envoi de la requête depuis le serveur vers les pages et la réponse d'une des pages vers le serveur web afin de pouvoir assurer une qualité de service plus précise.

6.3.4 Évaluation

Le fait d'avoir un support commun pour la communication entre les moteurs de raisonnement capables d'adapter les différents niveaux nous permet de concevoir de l'adaptation multi-niveaux.

Au travers de cette expérimentation, nous démontrons que le modèle de configuration offre un certain niveau d'abstraction pour faciliter l'adaptation multi-niveaux que ce soit en tenant compte des informations des différents niveaux ou en faisant collaborer les moteurs de raisonnement.

Pour cela, il est bien entendu nécessaire d'utiliser le modèle de configuration pour la définition des adaptations. Grâce au langage *KevScript*, la définition de reconfiguration nécessite peu de ligne de code (173 lignes de code).

De plus, grâce à la séparation des préoccupations, un moteur de raisonnement se concentre seulement sur la définition des reconfigurations et n'a pas besoin de définir l'implantation de ces reconfigurations. Le modèle de construction des différents types de nœud fournit lui aussi

Listing 6.13 – Définition du gestionnaire d'élasticité

```
@ComponentType
@DictionaryType({
  @DictionaryAttribute(name = "nbNodes", defaultValue="10", optional=true},
  @DictionaryAttribute(name = "maxPercentageCPULoad", defaultValue="95", optional=true}
})
class ElasticPaaSManager extends AbstractComponentType implements ModeListener {
  // this method is periodically executed
  def cronTask() {
    val model = getModelService.getLastModel
    // get the number of PaaS node
    val paasNodes = model.getNodes.filter(n => KloudModelHelper.isPaaSNode(model, n.getName))
    // look at the CPU load and if the load is larger than maxPercentageCPULoad
    val overloadedPaaSNodes = detectOverHead(paasNodes,
          Integer.parseInt(getDictionary.get("maxPercentageCPULoad")))
    if (overloadedPaaSNodes.size > 0) {
      overloadedPaaSNodes.foreach {
        currentNode =>
        val nbNodes: Int = model.getNodes.size
        val nodes: List[ContainerNode] = model.getNodes
        val nodeName = "node" + nbNodes + 1
        if (currentNode.getComponents.size > 1 && currentNode.getComponents.find(c =>
            c.getTypeDefinition.getName == "WebServer").isDefined &&
          currentNode.getComponents.find(c => c.getTypeDefinition.getName == "AbstractPage").isDefined) {
          // if all the WebPages are hosted in the same node than the WebServer, we migrate the component to
              another node
          kengine.addVariable("nodeName", nodeName)
          kengine.addVariable("currentNodeName", nodes.get(0).getName)
          kengine append "addNode {nodeName} : PJavaSENode"
          currentNode.getComponents.filter(c => c.getTypeDefinition.getName == "AbstractPage").foreach {
            component =>
            kengine.addVariable("componentName", component.getName)
            kengine append "moveComponent {componentName}@{currentNodeName} => {nodeName}"
          }
        } else if (currentNode.getComponents.size > 1 && currentNode.getComponents.find(c =>
            c.getTypeDefinition.getName == "WebServer").isEmpty) {
          // if the WebPages are not hosted in the same node than the WebServer, we create a new node for one
              of the WebPages
          kengine.addVariable("nodeName", nodeName)
          kengine append "addNode {nodeName} : JavaSENode"
          kengine append "addComponent {componentName}@{nodeName} : HelloWorldPage"
          kengine.addVariable("componentName", currentNode.getComponents.filter(c =>
              c.getTypeDefinition.getName == "AbstractPage").get(0).getName)
          kengine append "moveComponent {componentName}@{currentNodeName} => {nodeName}"
        } else if (currentNode.getComponents.size == 1 && currentNode.getComponents.find(c =>
            c.getTypeDefinition.getName == "WebServer").isEmpty) {
          // If each WebPage is hosted in its own node, we duplicate the WebPage that is on the overloaded node
          kengine.addVariable("nodeName", nodeName)
          kengine append "addNode {nodeName} : JavaSENode"
          kengine append "addComponent {componentName}@{nodeName} : HelloWorldPage"
          nodes.find(n => n.getName == getNodeName) match {
            case None =>
            case Some(node) =>
              if (node.getComponents.size == 1) {
                // copy all component parameters on the new replica
                KloudModelHelper.cloneComponent(model, node.getComponents.get(0), nodeName)
              }
          }
        }
      }
    }
  }
}
```

différens niveau d'abstraction permettant de créer des gestionnaires d'élasticité fonctionnant sur différents types de Cloud.

6.4 Synthèse

Ce chapitre a montré que l'utilisation du modèle de configuration proposé par Kevoree permet non seulement de définir des applications, des nœuds de plate-forme et des nœuds d'infrastructure mais qu'il permet aussi de définir des gestionnaires spécialisés capables de tenir compte de l'ensemble des informations fournies par les différents types et instances de nœuds, composants, canaux de communications et groupes. Ce sont ces gestionnaires spécialisés qui permettent de définir les spécificités de la plate-forme ou de l'infrastructure correspondante et permettent une réutilisation des types de nœuds peu importe la solution de plate-forme ou d'infrastructure que nous souhaitons développer. De plus, nous avons montré que l'impact de notre abstraction sur la gestion d'une infrastructure ou d'une plate-forme est négligeable par rapport au temps nécessaire à la création ou à l'arrêt des machines virtuelles ou dans notre cas d'étude d'espaces utilisateurs virtualisés. De même, nous avons montré que l'impact de l'utilisation mémoire de notre modèle reste acceptable. Enfin nous avons montré que l'utilisation de notre modèle de configuration permet de définir des systèmes d'adaptation capables d'utiliser les informations des différents niveaux pour effectuer de l'adaptation efficace. Ces systèmes d'adaptation sont aussi capables de collaborer grâce au modèle et ainsi assurer la cohérence de l'adaptation entre les niveaux ou encore définir des adaptations multi-niveaux.

Chapitre 7

Models@runtime pour les objets connectés

Ce chapitre présente un ensemble d'expérimentations pour qualifier l'utilisation d'un modèle de configuration à l'exécution dans le cadre de l'internet des Objets dans lequel les ressources de calculs peuvent être très limitées. Ces expérimentations ont été menées dans le cadre de la thèse de François Fouquet. Elles ont pour but de quantifier i) le temps moyen d'une reconfiguration (temps pendant lequel le système est à l'arrêt), ii) le surcoût en mémoire volatile et persistante lié à l'utilisation du modèle de configuration à l'exécution, iii) le délai pour redémarrer un équipement en cas de panne ou de reprogrammation complète de l'équipement.

Sommaire

7.1 Besoins spécifiques des systèmes adaptatifs contraints **104**
7.2 Capacité des micro-contrôleurs vis-à-vis des niveaux d'adaptation Kevoree **105**
7.3 Implantation d'un nœud Arduino Kevoree **106**
7.4 Validation expérimentale sur micro-contrôleur **107**
7.4.1 Axes d'évaluation 107
7.4.2 Protocole expérimental général 107
7.5 Expérimentation 7 : Downtime **108**
7.5.1 Configuration expérimentale 108
7.5.2 Limites de validité expérimentale 109
7.5.3 Resultats et analyse expérimentale 110
7.5.4 Extension expérimentale pour connaitre l'impact du type de mémoire. 111
7.6 Expérimentation 8 : combien d'instances déployables en mémoire volatile ? **112**
7.6.1 Protocole expérimental 112
7.6.2 Limites de validité expérimentale 112
7.6.3 Résultats expérimentaux et analyse 112
7.7 Expérimentation 9 : combien de reconfigurations successives? **113**
7.7.1 Limites de validité expérimentale 114
7.7.2 Résultats et analyse 114
7.8 Expérimentation 10 : Délai de redémarrage **114**
7.8.1 Limites de validité expérimentale 114

7.8.2 Résultats expérimentaux et analyse 115
7.9 Comparatif vis-à-vis d'un micro-logiciel non généré **116**
7.10 Conclusion vis-à-vis des axes d'évaluation **116**

L'introduction d'une abstraction se traduit généralement par un coût au niveau de la plate-forme d'exécution. Ce coût peut modifier le temps de réaction de la plate-forme face à une demande d'adaptation ou augmenter la quantité de mémoire utilisée. Ce coût peut être rédhibitoire pour un usage sur des nœuds dont la puissance de calcul ou la quantité de mémoire est trop faible. Si tel était le cas, ce constat serait contradictoire avec la capacité du modèle Kevoree à gérer l'hétérogénéité. Le but de cette deuxième expérimentation consiste à valider l'usage d'une approche de M@R sur des nœuds hétérogènes comprenant de fait des nœuds de plus faible puissance. L'objectif de cette section de validation est d'évaluer la viabilité de cet usage sur un cas limite d'usage du M@R avec l'adaptation sur des environnements fortement contraints.

Ce chapitre évalue ce point en prenant l'exemple des nœuds capteurs possédant des caractéristiques représentatives de ce domaine en privilégiant les approches matérielles open-source du domaine. Ces nœuds basse consommation et basse puissance sont le souvent utilisés en embarqué, par exemple dans le cas présent dans un équipement de sécurité incendie. Les résultats de cette section sont alors généralisables à des périphériques similaires et plus puissants.

Cette section détaille les attentes 7.1 d'un nœud embarqué Kevoree avant d'en extraire les métriques permettant une évaluation du coût introduit par une boucle autonomique [LMD13] utilisant une approche de modèle à l'exécution. L'évaluation expérimentale proposée effectue une série de mesures directement sur micro-contrôleur afin de connaître les limites d'utilisation de la solution.

7.1 Besoins spécifiques des systèmes adaptatifs contraints

Les nœuds d'exécution des objets connectés imposent de nouvelles contraintes vis-à-vis des nœuds plus *conventionnelles*. En effet, outre leur capacité plus restreinte ces périphériques sont utilisés sur des usages et avec des technologies qui imposent de respecter certains délais pour les temps de réaction ou encore imposent de réduire les écritures sur la mémoire pour allonger leur durée de vie. Les axes suivants représentent alors les défis de l'application du M@R et plus généralement ceux de n'importe quelle adaptation dynamique sur ces nœuds embarqués avec peu de ressources :

1. **Temps d'arrêt** : Les micro-contrôleurs hébergent un micro-logiciel qui contrôle souvent directement les périphériques physiques. Redémarrer ou bloquer ces micro-contrôleurs peut avoir de graves conséquences s'il s'agit de contrôler des périphériques critiques, ou des conséquences indésirables il s'agit de contrôler des équipements de confort. L'adaptation doit donc limiter ce temps d'arrêt (*downtime*) autant que possible.
2. **Utilisation de la mémoire volatile** (RAM) : L'allocation dynamique de mémoire est la brique élémentaire de l'adaptation dynamique. Les micro-contrôleurs exploitent le plus souvent quelques kilo-octets de mémoire vive, et cette limitation de taille interdit le plus souvent le stockage de plusieurs configurations en mémoire de manière simultanée.
3. **Utilisation de la mémoire persistante** : L'utilisation de la mémoire persistante est nécessaire pour garantir que le processus d'adaptation assure des modifications transactionnelles, pour ainsi assurer le recouvrement de l'état du micro-contrôleur en cas de redémarrage sur erreur. L'EEPROM est une mémoire embarquée directement dans les micro-contrôleurs, le plus souvent avec une taille très limitée. Ce type de mémoire a une durée de vie limitée en terme de nombre d'opérations d'écriture. De manière similaire

aux disques de type SSD [APW+08], les écritures dans une EEPROM doivent être distribuées sur les zones mémoires pour optimiser la durée de vie globale. L'utilisation de cette mémoire doit donc être limitée par la couche M@R pour assurer une pérennité du périphérique.

4. **Persistance d'état** : La capacité de recouvrement d'état est critique pour un système embarqué, qui est sujet à des pannes fréquentes (par exemple suite à une perte d'alimentation). Les micro-contrôleurs doivent alors redémarrer et restaurer la configuration précédente rapidement pour reprendre le fonctionnement nominal, et ceci en prenant en compte de nombreuses modifications d'architecture successives. Cette propriété est assurée par les périphériques car ils démarrent à partir d'une mémoire persistante. La mise à jour de leur micro-logiciel se fait *via* une opération de *flash* (écriture dans la mémoire) qui couvre ce besoin de persistance. L'adaptation dynamique doit donc respecter cette fonctionnalité.

D'une manière générale, les applications à base d'objets connectés exploitent un large ensemble de nœuds autonomes avec de fortes contraintes énergétiques, ce qui incite à choisir des plates-formes peu chères et capables de fonctionner sur de longues périodes sans acte de maintenance (par exemple pour changer une batterie). Les micro-contrôleurs de type AVR [1] prennent en charge ces besoins pour les raisons suivantes :

1. Leur base d'architecture 8 bits de type Harvard est simplifiée et robuste, rendant le temps d'exécution prévisible : un micro-contrôleur peut opérer dans une large bande de température (typiquement de -40 à 85 degrés Celsius), d'humidité, et d'alimentation électrique ; un tel micro-contrôleur a un nombre fixe de cycles pour exécuter une opération.
2. Leurs besoins énergétiques (et la chaleur dégagée) sont très faibles : un micro-contrôleur 8 bits fonctionnant à 32 kHz consomme typiquement 0,05 W (moins de 0.5 W à 1 MHz). Ils peuvent de fait fonctionner pendant de longues périodes sur batterie.
3. Leur architecture simplifiée permet la production de masse, faisant des micro-contrôleurs des nœuds très peu chers, capables d'être déployés en grand nombre en *cluster*.

Les micro-contrôleurs répondent donc aux besoins des objets connectés, ce qui explique, bien évidemment, leur usage intensif dans ce domaine et dans celui voisin des réseaux de capteurs. Si les AVR sont des processeurs relativement anciens au moment de cette évaluation, ils évoluent vers des architectures autour des produits de type ARM Cortex-M [2] plus puissants en tous points. Le choix des processeurs AVR pour cette évaluation correspond au pire cas en terme de puissance de processeur. Par conséquent, les résultats de viabilité seront de fait généralisables à de nouvelles architectures.

7.2 Capacité des micro-contrôleurs vis-à-vis des niveaux d'adaptation Kevoree

La modélisation des nœuds de type micro-contrôleur donne lieu à la création d'un *NodeType* Kevoree .

Les qualités de mémoire embarquée qui apporte la robustesse à ce type de noeud est également leur faiblesse. En effet toute modification de micro-logiciel embarqué nécessite l'écriture complète en mémoire *flash* du nouveau programme. C'est le cas par exemple lorsqu'on modifie une définition de type. Toute la difficulté est alors de trouver un compromis pour l'équivalence

1. http ://www.atmel.com/Images/doc2545.pdf
2. http ://www.arm.com/products/processors/cortex-m/index.php

avec les concepts de Kevoree , entre la flexibilité offerte et le coût d'exploitation. Il faut alors trouver des solutions légères pour les quatre niveaux d'adaptation suivant leurs taux d'usage.

L'adaptation dynamique de *TypeDefinition* correspond à un ajout de fonctionnalité qui correspond par exemple à l'ajout d'un capteur physique. La mise à jour architecturale ou paramétrique correspond à un changement de configuration (par exemple un ajout de liaison) ou de contexte (par exemple un changement de paramètre). Les changements de *TypeDefinition* sont bien moins réguliers dans les cas d'usage car ils nécessitent une intervention physique. La solution d'adaptation envisagée doit donc assurer un coût inférieur pour les adaptations architecturales, quitte à diminuer les performances des mises à jour de conception continue.

7.3 Implantation d'un nœud Arduino Kevoree

Le micro-contrôleur choisi pour l'expérience de validation est issu de l'environnement Arduino. Arduino [3] est un concept de plate-forme matérielle et logicielle *open source* pour la réalisation de prototype fondé sur un processeur AVR 8-bits. Connecté à des capteurs et actuateurs physiques ce type de plate-forme répond aux besoins des objets connectés et les résultats qui en découlent sont généralisables à d'autres familles de processeurs tels que les PIC ou les ARM.

L'implantation d'un tel nœud repose sur une équivalence entre les concepts Kevoree et les concepts manipulables dans ce type d'environnement. Ainsi les *TypeDefinition* reposent sur une implantation des composants en C. Le concept d'*Instance* de Kevoree repose sur une création dynamique de zone mémoire correspondant à ces structures tandis que les liaisons dynamiques entre *Composants* et *Channels* reposent sur des liaisons par tableaux de référence dynamique C. Les échanges de données suivent le modèle Kevoree : les ports sont implantés sous la forme de file d'attente de type FIFO, pour protéger les composants des accès externes. Les systèmes de base des micro-contrôleurs étant trop petits pour héberger une couche de système opérationnel (OS), aucun mécanisme d'ordonnancement n'est prévu. Pour pallier ce manque, un ordonnanceur est introduit pour gérer les files de messages et ainsi prioriser les instances. Cet ordonnanceur est en charge de l'équilibre entre les exécutions périodiques requises par certains composants et la taille des files d'attente. Il vise ainsi à garder le système sain et éviter un phénomène de surcharge localisée.

Flasher le microcontrôleur pour les évolutions majeures

Un microcontrôleur peut être programmé une fois pour toutes afin qu'il effectue une ou des tâches précises pour une ou des applications précises. Mais les microcontrôleur récents peuvent être reprogrammés et ceci grâce à leur mémoire reprogrammable de type FLASH (d'où le terme flasher quelque chose). Remplacer la totalité du micro-logiciel et redémarrer le périphérique est donc une implantation possible de l'adaptation dynamique pour le nœud microcontrôleur. Si cette technique est raisonnable pour des périphériques connectés en filaire ou pour des maintenances physiquement connectées (opérateur physiquement présent) elle prend néanmoins plusieurs secondes. Cependant cette technique est problématique pour les périphériques non accessibles physiquement ou ayant des contraintes forçant une mise à jour en moins d'une seconde. Envoyer un micro-logiciel à travers les airs est une opération hasardeuse : en effet les micro-logiciels contiennent un grand nombre de données, et la gestion des erreurs de communication implique d'y ajouter un certain nombre d'informations supplémentaires pour valider la réception et gérer les erreurs. En pratique ceci allonge encore le temps de flashage

3. http ://www.arduino.cc

au travers d'un réseau non filaire, rendant l'opération délicate et nécessitant un matériel et un micro-logiciel de démarrage dédié.

Dans l'approche proposée, la reprogrammation complète de la mémoire flash n'est requise que pour toute modification de *type définition*, par exemple lorsqu'un nouveau composant est ajouté à la librairie. De ce fait les micro-contrôleurs conçus en langage C n'offrent pas la même flexibilité que des environnements Java tels que les nœuds OSGi, car ils ne sont pas capables d'incorporation et de chargement de classe de type à chaud. Ceci est typiquement nécessaire pour une configuration initiale où les types envisagés sont déployés et pour les évolutions majeures du système (par exemple. pour prendre en compte un type non visible au déploiement initial). Pour tous les autres cas l'approche proposée permet de faire une reconfiguration des instances en mettant à jour uniquement une partie de la mémoire programme, rendant l'opération beaucoup plus rapide et garantissant l'encapsulation de l'approche Kevoree puisque le micro-contrôleur reste maître de sa plate-forme.

7.4 Validation expérimentale sur micro-contrôleur

Pour valider la viabilité de la solution M@R sur des environnements aussi contraints, une série d'évaluations a été réalisée sur des micro-contrôleurs réels.

7.4.1 Axes d'évaluation

Chacune de ces évaluations cherche à quantifier le surcoût sur chacun des axes détaillés dans les motivations. Ainsi l'évaluation globale porte sur les métriques suivantes :

- **Downtime** : temps d'adaptation globalement pris par le micro-contrôleur pour changer d'état et appliquer un nouveau modèle, incluant le temps d'envoi du nouveau modèle. Cette métrique définit le temps pris par le micro-contrôleur mono-tâche pour la gestion de son propre état et non pour la réalisation de sa tâche applicative (lecture de capteur par exemple). Ce délai est donc perdu d'un point de vue application métier.
- **Utilisation de la mémoire volatile (RAM)** : taux d'utilisation de la mémoire RAM dédiée à l'allocation dynamique d'instances Kevoree (composants, *channels*, etc). Cette métrique permet d'anticiper le nombre maximum d'instances qu'un nœud Arduino peut contenir.
- **Utilisation de la mémoire persistante** : taux d'utilisation de mémoire persistante pour stocker les états du nœud. Ce taux d'occupation dans le temps donne également de manière transitive une métrique qui évalue le taux de répartition du stockage et donc de l'usure de la mémoire. Par exemple la mémoire persistante embarquée dans les AVR 8 bits offre un nombre limité d'écriture certifié pour chaque octet et donc limite dans le temps le stockage possible de l'historique des états. Ce stockage est nécessaire pour la résilience de chaque reconfiguration.
- **Temps de redémarrage et de récupération** : temps pris par le micro-contrôleur pour restaurer son précédent modèle et son état après un crash, par exemple dû à une coupure de courant.

7.4.2 Protocole expérimental général

L'ensemble des expériences a été réalisé sur l'implantation de référence du noeud Kevoree pour Arduino. Le micro-contrôleur utilisé est un ATMEL AVR 328P. Ce processeur embarque 32 KB de mémoire flash pour stocker le micro-logiciel ainsi que 2 KB de mémoire RAM et 1 KB de mémoire persistante de type EEPROM. Une mémoire flash additionnelle (microSD connecté

via un bus SPI) a été ajoutée pour certaines expériences afin de mesurer l'impact du type de mémoire sur les résultats.

Pour simuler des changements de configuration, un ensemble de modèles représentant les différents états sont créés. Ces modèles sont issus d'un cas d'étude de *smart building* tel que détaillé dans la publication à la conférence CBSE'12 [FMF+12]. De manière schématique ces modèles représentent un nœud micro-contrôleur qui pilote 5 capteurs physiques et qui peut communiquer avec un nœud passerelle souvent plus puissant à l'étage d'un bâtiment. Les modèles représentent différents usages de ce capteurs, un cas d'urgence où les capteurs échantillonnent beaucoup de valeurs et pilotent une alarme, un cas de confort où le capteur pilote une lumière et un chauffage, etc. Les modèles utilisés dans cette expérimentation diffèrent d'environ dix instances Kevoree , la figure 7.1 illustre un de ces modèles.

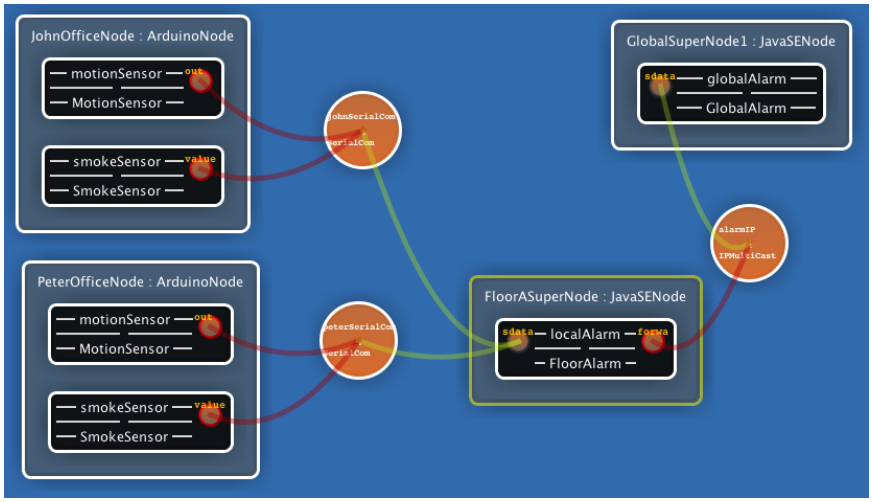


FIGURE 7.1 – Illustration modèle Kevoree de SmartBuidling

7.5 Expérimentation 7 : Downtime, combien de temps les adaptations bloquent-elles la logique métier ?

7.5.1 Configuration expérimentale

Dans cette expérience, cinq modèles tirés du cas d'étude *smart building* sont sélectionnés ; ils correspondent à des changements de configuration des capteurs d'une pièce pour un usage diurne ou nocturne. Dans ces modèles, quatre nœuds sont présents, comprenant chacun de 0 à 10 *Instances* chacun. Chaque *Instance* utilise un *TypeDefinition* et comporte en moyenne environ 30 lignes de code.

Dans une première étape, le déploiement initial du premier état installe un modèle contenant tous les *TypeDefinition* exploités dans cette expérience. Cette mise à jour majeure est faite par une opération d'écriture de la mémoire flash pour charger le programme du micro-contrôleur, précédée d'une étape de génération de code et compilation. Dans sa version expérimentale le nœud Kevoree Arduino introduit des sondes dans le code généré pour mesurer le temps d'inactivité (*downtime*) et l'usage de la mémoire (EEPROM et SDRAM). Après cette étape initiale, toutes les 100 ms un nouveau modèle est choisi de façon aléatoire et est synchronisé

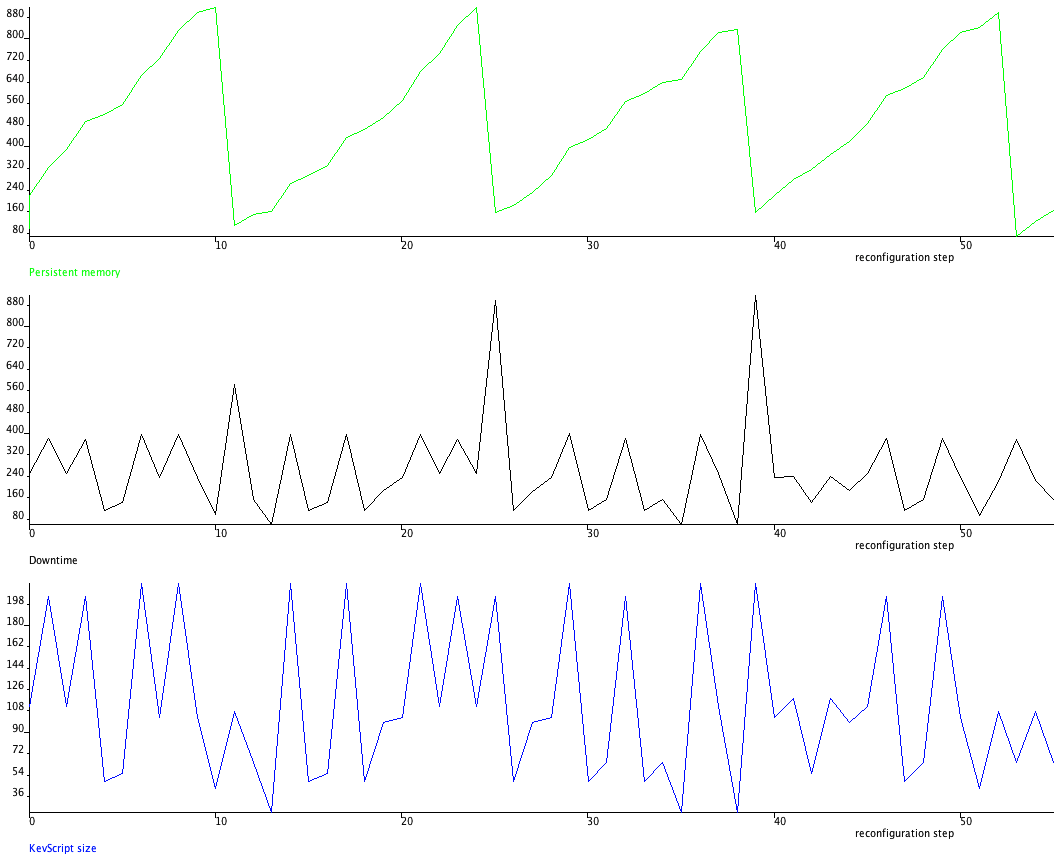


FIGURE 7.2 – Résultats expérimentaux bruts

avec le micro-contrôleur, remplaçant ainsi la précédente configuration. Au total 500 changements d'état sont effectués consécutivement. La figure 7.2 représente les graphes des données brutes collectées lors de ce test. Le tracé du haut illustre l'usage de la RAM et démontre sa constance. Le second tracé illustre le temps de *downtime* par reconfiguration. Le troisième et dernier tracé illustre la taille de script de configuration pour piloter le nœud à distance.

7.5.2 Limites de validité expérimentale

Interne

La régularité des mises à jour de modèles (toutes les 100 ms) ainsi que le caractère synchrone du déploiement par le processus externe introduit un premier biais de validité interne. En effet, ce déploiement synchrone régule les émissions de modèles en fonction de la capacité maximale de traitement du micro-contrôleur. Le délai de 100 ms est volontairement en dessous des valeurs nécessaires pour le cas d'usage de l'Internet des objets, cependant pour des valeurs inférieures il serait alors nécessaire pour le nœud externe de faire tampon et de cumuler les mises à jour avant envoi au micro-contrôleur. Ce traitement introduirait un coût en temps qui serait dépendant de la capacité de calcul.

Le temps de prise de contrôle du micro-contrôleur est également dépendant de la rapidité

d'interruption matérielle offerte par le support de transfert, ici une liaison série.

Externe

Le temps de déploiement initial pour la première configuration inclut un temps de compilation du micro-logiciel. Ce dernier est dépendant de la puissance du nœud de soutien connecté au micro-contrôleur.

Les délais présentés ici dépendent également de la vitesse de communication de la liaison entre le nœud de soutien et le micro-contrôleur. Réalisé ici avec une liaison série cadencée à 9600 bauds, ceci représente une valeur inférieure à la capacité moyenne des puces de communications tel que Xbee. Cependant, dans le cas des transmissions radio, la valeur de débit varie en fonction de la distance entre l'émetteur et le récepteur. Le canal expérimental filaire choisi prévient la plupart de erreurs de communication, le traitement de ces dernières inclut inévitablement un coût en temps pour l'émission et pour le micro-contrôleur pour valider la réception.

7.5.3 Resultats et analyse expérimentale

Le déploiement initial et plus généralement les mises à jour majeures s'avèrent très coûteuses : le temps de *downtime* pour cette étape est de 12,208 s. Cette valeur importante s'explique notamment par le temps pris par le transfert du micro-logiciel mais également par le temps pris par le micro-contrôleur pour le redémarrage. Cette valeur varie dans une fourchette de +/- 2 secondes.

Les résultats de cette première expérience mettent en lumière la corrélation logique entre la taille des scripts de reconfiguration et les temps mesurés de *downtime* : le taux de corrélation de Spearman [4] observé entre la taille des scripts et les temps de *downtime* est supérieur à 0,9. De plus, l'étape de compression exploitée pour réduire l'historique stocké dans l'EEPROMa également un impact sur le *downtime*. On observe alors que les déclenchements d'exécution de cette tâche sont directement corrélés avec les plus fortes valeurs de *downtime*.

Après 500 cycles de reconfiguration, on observe les valeurs maximales et moyennes suivantes :

— *downtime* minimum de 58 ms, 210 (c.-à-d. 12208 / 58) fois plus rapide que le flashage initial ;

— *downtime* maximum de 916 ms, 14 (c.-à-d. 12208 / 916) fois plus rapide que le flashage initial ;

— *downtime* moyen de 235 ms, 52 (c.-à-d. 12208 / 235) fois plus rapide que le flashage initial.

Voici la représentation en graphe et table de percentiles pour une meilleure analyse de la répartition des valeurs de *downtime*.

Percentile(%)	0	5	25	50	75	95	100
Downtime (ms)	58	59	139	221	248	398	916

La figure 7.3 synthétise le résultat de cette expérience. Cette figure est également utilisée dans les expérimentations suivantes. Pour cette expérimentation, seul le premier graphe est exploité.

La même figure 7.3 montre clairement que la répartition de valeurs de *downtime* se regroupe autour de 220 ms. 95% de ces valeurs sont inférieures à 400 ms et 75% sont inférieures à 250 ms. Seulement 5% de ces valeurs sont au dessus de la barre des 400 ms, ceci s'explique par l'étape de compression de l'EEPROM. La compression en mode juste à temps permet bien de limiter le nombre de pics de *downtime* et elle maintient les valeurs autour de 200 ms.

4. http://fr.wikipedia.org/wiki/Corrélation_de_Spearman

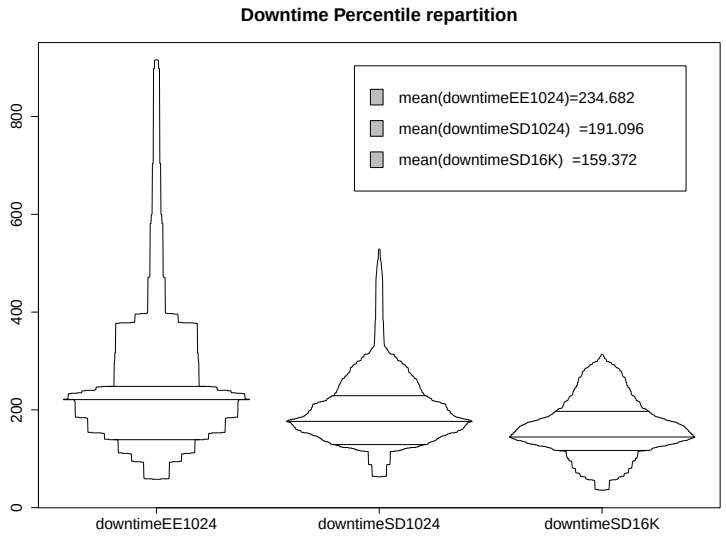


FIGURE 7.3 – Distribution percentile du temps de downtime(en ms)

L'étude suivante de Tapani *et al* [Kos08] aborde justement l'identification d'une valeur seuil pour un périphérique fournissant un service à un humain. Cette étude conclut : « *Delays base on the human nervous system [. . .] It takes 190 - 215ms for light stimuli to be processed.* »La valeur seuil est donc particulièrement intéressante car elle correspond au temps de réaction perceptible par un humain. Une adaptation qui reste proche ou en dessous de ce seuil sera perçue comme immédiate par un humain s'il en est l'origine. Les plus grandes valeurs de *downtime* sont systématiquement liées à une réduction de la mémoire EEPROM utilisée (routine de compression). On observe 32 compressions pendant les 500 configurations, soit 6.4% des reconfigurations nécessitant une compression pour être stockées. La valeur maximale de ces 32 pics est de 916 ms, la valeur minimale est de 218 ms et la moyenne est de 580,815 ms.

Les sondes injectées inspectent l'usage de la RAM pendant l'expérience, aucune fuite mémoire ni fragmentation n'est observée. Bien que le taux d'usage de la RAM soit stable il est nécessaire que le surcoût introduit par le *framework* d'instanciation dynamique ne limite pas la création d'un nombre réaliste d'instances. La prochaine expérimentation aborde ce point.

7.5.4 Extension expérimentale pour connaitre l'impact du type de mémoire.

Les mêmes 500 itérations ont été réalisées en remplaçant la mémoire EEPROM par une mémoire externe de type *flash* de 2 GB (carte SD) connectée *via* un bus SPI. Cette expérience est répétée deux fois, avec une taille de stockage de 1 kB (de manière identique à l'EEPROM) puis 16 kB. Les résultats sont montrés dans le tableau suivant :

Percentile(%)	0	5	25	50	75	95	100
Downtime SD 1K(ms)	63	88	129	176	229	324	529
Downtime SD 16K(ms)	35	56	117	145	197	297	314

La mémoire *flash* a un temps d'initialisation plus long que l'EEPROM, ce qui explique que la plus petite valeur de *downtime* de l'expérience soit plus grande que la plus grande de celle de l'EEPROM. Cependant la rapidité d'écriture et de transfert est plus grande, ce qui conduit

à une homogénéisation des valeurs de *downtime* qui passe sous la barre des 200 ms dans la plupart des cas. On peut également noter qu'une valeur plus grande de mémoire persistante (16 kB) lisse les pics et fait légèrement baisser la valeur moyenne. Cependant la distribution des valeurs reste similaire. Les performances de la carte mémoire utilisée peuvent légèrement faire varier ces résultats.

7.6 Expérimentation 8 : Utilisation de la mémoire volatile, combien d'instances déployables ?

L'objectif de cette expérience est d'évaluer le surcoût en terme d'utilisation de mémoire volatile et ainsi valider la capacité restante vis-à-vis du déploiement des composants métiers.

7.6.1 Protocole expérimental

Pour mesurer cette capacité et donc le nombre maximal d'instances déployables, le protocole expérimental consiste à déployer une configuration initiale puis à l'étendre de façon cyclique. Cette configuration initiale du cas d'usage *smart building* comprend 3 instances : un composant *timer*, un composant gérant un interrupteur et un canal de communication entre les deux. Toutes les 100 ms ce modèle est étendu en ajoutant un nouveau composant interrupteur et en le liant au canal de communication existant. Des sondes sont injectées pour mesurer la mémoire SDRAM.

7.6.2 Limites de validité expérimentale

Interne

Le cas d'étude *SmartBuilding* exploite des composants de type capteur dont les tailles mémoires nécessaires pour leurs fonctionnement sont sensiblement identiques. Il en résulte une fragmentation de la mémoire du micro-contrôleur relativement faible. La fragmentation est inhérente à tous les processus d'allocation de mémoire lorsque les tailles de composants sont hétérogènes ; ceci n'est pas évalué ici.

Externe

Les résultats de cette expérience sont directement liés à la taille mémoire des composants choisis pour l'expérimentation. En effet la mémoire dynamique nécessaire pour chaque composant est elle-même liée à la complexité de son calcul de discrétisation de la valeur physique. Le type de capteur à mesurer influe donc sur les valeurs de ces résultats.

7.6.3 Résultats expérimentaux et analyse

L'expérience est menée jusqu'au remplissage total de la mémoire du capteur, interdisant tout nouveau déploiement. Deux types de micro-contrôleurs sont utilisés, un AVR ATMEL modèle 328P et un 2560, les résultats sont synthétisés dans le graphique suivant 7.4.

Les résultats pour le micro-contrôleur 328P montre que la mémoire SDRAM est pleine après environ 22 cycles de reconfiguration, ce qui veut dire que le micro-contrôleur a été capable de déployer 25 instances de composants (3 initiales et 22 additionelles). En pratique le nombre de périphériques géré par un micro-contrôleur est souvent proche du nombre de broches qu'il peut gérer. En supplément, un ou plusieurs composants additionnels sont nécessaires pour orchestrer les périphériques et faire les calculs dépendants. Dans notre cas on peut alors gérer 25 instances

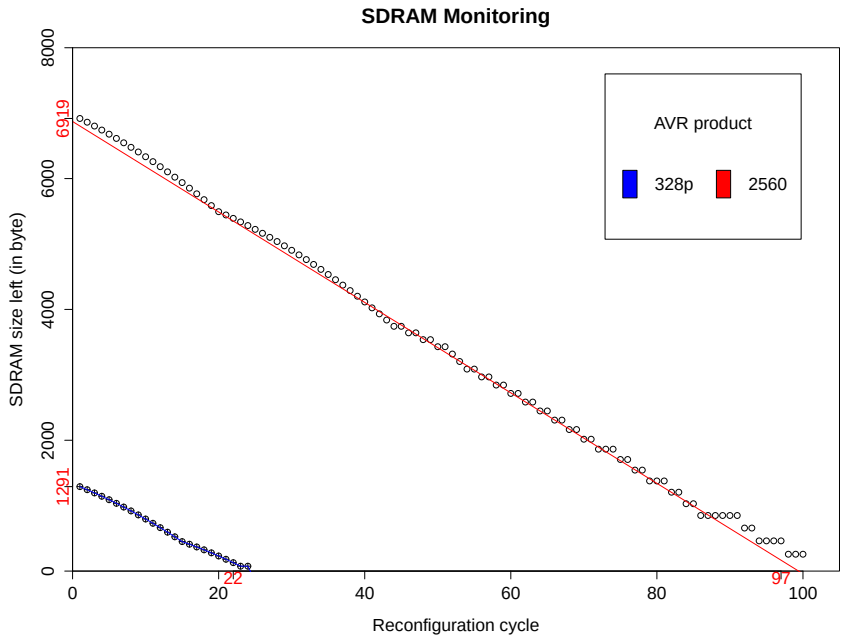


FIGURE 7.4 – Résultat expérimental sur la capacité mémoire volatile

qui comprennent des composants d'orchestration et de gestion de périphérique comparativement aux 22 broches du micro-contrôleur de test.

Dans une deuxième expérience, un micro-contrôleur ATMEL 2560 est exploité avec 4 KB de mémoire. Les résultats convergent vers une valeur de 100 instances déployées, l'amélioration de 300% est donc cohérente avec l'augmentation de capacité. Encore une fois le nombre d'instances est plus grand que le nombre de broches (80) que le micro-contrôleur peut gérer. Malgré un surcoût mémoire, l'approche est donc compatible avec l'état de la pratique.

7.7 Expérimentation 9 : Utilisation de la mémoire persistante, combien de reconfigurations successives peuvent être déployées ?

Les micro-contrôleurs ont des capacités limitées en terme d'écriture. Ces contraintes venues du matériel exploité dans le monde de l'embarqué peuvent être antinomiques avec l'usage de l'historique nécessaire pour la persistance du Models@Runtime. L'expérience suivante vise donc à valider cet usage dans le cas d'usage *smart building*. Le protocole expérimental est le même que pour la première expérience (Section 7.5). Cinq modèles du cas d'usage standard sont déployés de manière cyclique. Des sondes sont injectées afin de mesurer les écritures dans la mémoire persistante. Dans cette première expérience, la mémoire embarquée EEPROM du micro-contrôleur 328P est utilisée, 500 cycles de déploiement sont effectués.

7.7.1 Limites de validité expérimentale

Interne

Les écarts types (du nombre d'instances modifiées) entre chaque configuration influent sur le nombre total de configurations à sauver sur le support persistant.

Externe

Les résultats de cette expérience sont directement liés à la taille mémoire persistante nécessaire pour le stockage d'un composant. Cette taille par composant est elle-même directement liée aux nombres et aux types de paramètres dynamiques de celui-ci. Les composants choisis pour l'expérience sont représentatifs du cas d'étude et comprennent entre 1 et 4 paramètres de type entier.

7.7.2 Résultats et analyse

On observe 32 pics de compression de la mémoire EEPROM durant les 500 cycles soit en moyenne une compression tous les 15,625 changements de modèle. De manière analogue à la mémoire employée dans les *Solid State Disks* [APW+08], chaque opération d'écriture sur la mémoire EEPROM entraîne une usure. Cette usure doit être répartie de manière uniforme sur la mémoire pour éviter la perte prématurée de zone mémoire. Dans le pire des cas l'algorithme de compression effectue un cycle d'écriture sur chaque octet de la mémoire pour la compression d'un état initial. Chaque octet de ce type de mémoire est garanti pour 100 000 écritures [5]. Par extrapolation si on suppose que 100 reconfigurations peuvent être effectuées par jour (ce qui est bien plus que le cas d'usage envisagé), chaque octet de l'EEPROM sera écrit 6,4 fois par jour en moyenne. Dans ce cas d'usage la mémoire mettra 15 625 jours à atteindre son taux d'usage certifié, soit une durée de vie d'environ 43 années pour ce système piloté par le Model@Runtime.

7.8 Expérimentation 10 : Délai de redémarrage, combien de temps pour la récupération d'état ?

La sauvegarde d'état et sa restauration prennent un temps non négligeable lors du démarrage du capteur. La persistance de cet état est un besoin critique dans ce cas d'usage, car le capteur est souvent confronté à des pertes d'alimentation électrique. L'expérience cherche à montrer que la couche Models@Runtime permet un démarrage compatible avec les besoins du cas d'usage.

Le protocole expérimental général est ici ré-exploité sur 50 cycles de reconfiguration espacés de 2 secondes. Le micro-contrôleur est physiquement redémarré entre chaque reconfiguration. Une nouvelle sonde est ajoutée dans le micro-logiciel pour la mesure du temps de démarrage de la couche d'adaptation générée.

7.8.1 Limites de validité expérimentale

Interne

Le *framework* exploité (librairie Arduino) pour l'implantation de Kevoree peut avoir une influence négative sur les performances de lecture sur les mémoires externes.

5. http ://arduino.cc/en/Reference/EEPROMWrite

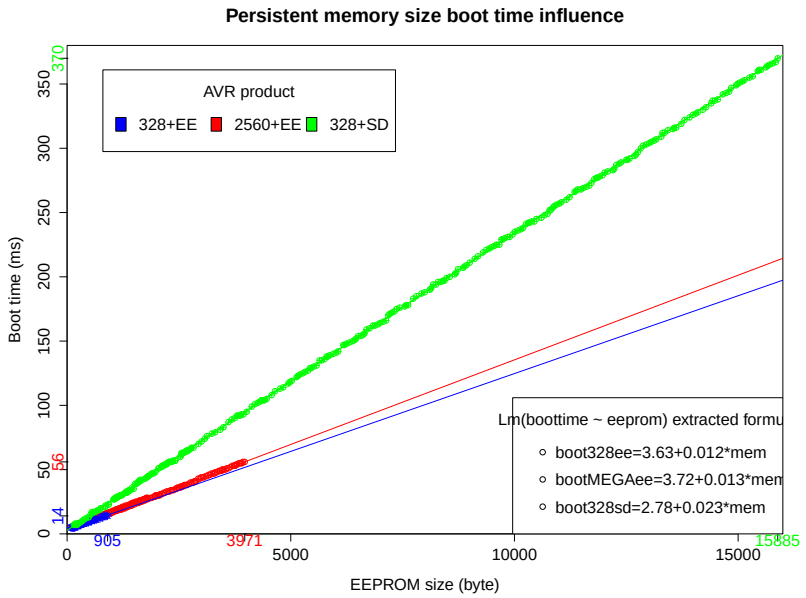


FIGURE 7.5 – Influence de la mémoire persistante sur le temps de démarrage

Externe

Les performances en lecture de la carte SD exploitée pour l'expérience peuvent influencer légèrement les résultats. Les performances de la carte utilisée (de type 4) correspondent cependant à la moyenne des performances de ce type de mémoire.

7.8.2 Résultats expérimentaux et analyse

Les premières mesures sont effectuées sur un micro-contrôleur 328P et une mémoire interne EEPROM. Ils sont représentés dans la figure 7.5 par la courbe verte (les autres tracés de la figure font partie de la deuxième série de mesures expliquée par la suite).

Ces résultats montrent que le pire cas avec ce processeur AVR et lorsque la mémoire EEPROM de 1 KB est pleine entraîne un temps de démarrage d'environ 15 ms. Dans les meilleurs mesures, lorsque la mémoire est relativement vide après une compression d'état, le temps de démarrage devient alors de 3 à 4 ms. Cette valeur seuil proche du pire cas est notamment explicable par la lenteur du lecteur de ce type de mémoire. Cependant le temps de démarrage reste largement acceptable dans ce cas d'usage car inférieur aux 200 ms décrits dans l'étude de Tapani *et al* [Kos08] perceptible pour un humain même dans le pire des cas. Il est même possible d'inférer que le temps de démarrage étant beaucoup plus court que le temps d'écriture, il est intéressant d'exploiter l'ensemble de la mémoire disponible.

Extension du protocole avec un nouveau type de mémoire

Une deuxième série de mesures est lancée en remplaçant la mémoire EEPROM par une mémoire de type *flash* de taille de 1 KB et 16 KB. Cette mémoire externe de type SD, connectée par un bus SPI entraîne alors un temps de démarrage plus long. On observe un coefficient de 0.012 pour l'EEPROM comparativement à un coefficient de 0.023 pour la mémoire SD. Ceci s'explique par les temps d'écriture et lecture ralentis par le bus. Le temps de démarrage est linéairement distribué en fonction de la taille allouée (1 ou 16 KB). Au-delà de cette limite le

temps de démarrage (360 ms) dépasse la valeur moyenne du temps de reconfiguration moyen et perd alors son intérêt.

7.9 Comparatif vis-à-vis d'un micro-logiciel non généré

Une dernière expérience permet de quantifier le surcoût de l'approche dynamique par rapport à un micro-logiciel statique écrit à la main par un humain. Pour cette quantification on mesure la taille mémoire volatile et programme pour un exemple *HelloWorld* en C écrit à la main et un autre généré. Les micro-logiciels résultants sont déployés sur un micro-contrôleur 328P. Après la séquence de démarrage la version C laisse 1842 octets disponibles, tandis que la version générée par Kevoree avec le *framework* dynamique laisse 1604 octets libres. Le surcoût représente donc 11% de la mémoire RAM total disponible (2048). La version C prend également 2.3 KB de mémoire programme et la version Kevoree 7.3 KB soit 15% des 32 KB disponibles. Le surcoût sur la plus petite puce de cette étude (et donc le cas le plus défavorable), se limite donc à moins de 15% des différentes mémoires disponibles, bien sûr ce taux diminue sur des puces plus puissantes.

7.10 Conclusion vis-à-vis des axes d'évaluation

Sur chacun des axes qui ont fait l'objet d'une expérimentation, l'introduction de la couche d'adaptation dynamique dirigée par le Model@Runtime s'est avéré compatible avec les pré-requis de l'embarqué et du cas d'usage. Bien évidement les seuils d'acceptabilité de temps d'arrêt ou de redémarrage sont à mettre en corrélation avec les cas d'usage. Ainsi si l'approche exploite ici un cas d'usage issu de l'informatique embarquée pour cette validation, celle-ci ne couvre néanmoins pas tous les cas d'usage critiques. Certains cas d'usage pourraient par exemple nécessiter des valeurs seuil inférieures aux résultats, cependant l'approche d'évaluation aux limites valide ici un cas d'usage fortement lié à l'Internet des Objets. Ce cas limite est donc compatible d'un point de vue logique et technique avec l'approche Model@Runtime, rendant possible l'intégration des objets connectés dans la modélisation d'un système adaptatif distribué et hétérogène.

Chapitre 8

Bénéfices liés à l'utilisation du models@runtime pour construire un système de surveillance dynamique et optimiste

Ce chapitre présente un ensemble d'expérimentations pour montrer les bénéfices liés à l'utilisation du models@runtime pour construire un système de surveillance dynamique et optimiste. Ces expérimentations ont été menées dans le cadre de la thèse d'Inti Y. Gonzalez-Herrera. Elles ont pour but de montrer les gains obtenus en termes de coût en mémoire et en temps d'exécution pour la mise en œuvre d'un système de surveillance. Nous montrons en particulier que la conservation du modèle de configuration à l'exécution permet de construire un système de surveillance dynamique et optimiste qui offre l'avantage d'être efficace tout en gardant un bon niveau de précision pour diagnostiquer un ensemble de composants défectueux.

Sommaire

8.1 Contexte **118**
8.2 Vue générale de l'approche Scapegoat **118**
8.2.1 Contrat de qualité de service 119
8.2.2 Un conteneur muni de mécanismes de surveillance dynamique . . . 120
8.2.3 Architecture de Scapegoat 122
8.3 Protocole expérimental commun **123**
8.3.1 Cas d'étude 123
8.3.2 Méthodologie de Mesure 124
8.4 Expérimentation 11 : Coût lié à l'instrumentation **124**
8.4.1 Protocole expérimental 124
8.4.2 Résultats expérimentaux et analyse 125
8.5 Expérimentation 12 : Comparaison du coût de la surveillance adaptative face à la surveillance pleine **126**
8.5.1 Protocole expérimental 126

8.5.2 Résultats expérimentaux et analyse 126
8.6 Expérimentation 13 : Coût lié à la commutation entre modes de surveillance **127**
8.6.1 Protocole expérimental 127
8.6.2 Résultats expérimentaux et analyse 128
8.7 Discussions **129**
8.7.1 Limites de validité expérimentale 129
8.7.2 Travaux en cours 130

8.1 Contexte

Comme évoqué précédemment dans l'expérimentation autour du parallélisme des tests (cf Section 6.2.2.1, le partage des ressources, la réservation et l'isolation est un enjeu important dans des environnements dynamiques et ouverts. En effet, dans le domaine de la téléphonie que ce soit dans le set-top box ou les smartphones, que ce soit dans le domaine de l'infrastructure de Clouds mais de plus en plus dans le domaine des objets connectés en tout genre, de nombreux *frameworks* supportent le déploiement continu et l'exécution parallèle de composants logiciels au sein du même système ou de la même machine virtuelle. C'est par exemple le cas en Java dans des frameworks comme OSGi. Dans la plupart de ces *frameworks*, l'isolation entre composants est limitée. Une version défectueuse de n'importe lequel des modules logiciels peut mettre en péril le système entier en consommant toutes les ressources disponibles.

Dans ce chapitre, nous présentons une approche, nommée Scapegoat, dans laquelle nous proposons un mécanisme de surveillance dynamique en vue de limiter le coût lié à la surveillance tout en offrant une capacité à détecter le module ou les modules défectueux de manière efficace. La surveillance est fondée sur des heuristiques permettant de soupçonner certains modules. Ces modules sont alors instrumentés pour permettre l'analyse plus profonde par le système de surveillance, mais uniquement à la demande. Les modules dit « de confiance » sont laissés intacts et s'exécutent normalement. Cette technique nous permet un mécanisme de surveillance à la demande qui diminue le coût global du système de surveillance.

8.2 Vue générale de l'approche Scapegoat

L'hypothèse du monde ouvert a permis la naissance de nombreux cadres de développement permettant le chargement dynamique de nouveaux modules sans interruption de service. Néanmoins, l'isolement entre les composants est souvent limité parce qu'ils sont exécutés au sein d'une même machine virtuelle. Dans ces environnements fortement imprévisibles, détecter le comportement irrégulier d'un module logiciel et maintenir le système dans un état cohérent et stable est une préoccupation importante qui entraîne généralement la mise en place de sonde et de monteur de surveillance su système en cours d'exécution.

De nombreux mécanismes de surveillance [FS04, KHW03, BH06] permettent de connaître les valeurs de paramètres de système, comme le temps d'exécution d'un composant, sa consommation mémoire ou le nombre d'entrée/sorties, ou le nombre d'appels vers ce composants. Le coûts de ces mécanismes de surveillance est souvent le principal problème qui gène leur utilisation en production. Ainsi, Binder *et al.* [BHMV09] montrent que le coût en terme de temps d'exécution pour un système finement surveillé dépasse généralement un facteur de 4.3. L'expérience suivante montre en outre que ce coût a tendance a augmenté en fonction de la complexité du système surveillé. Ce coût important limite souvent l'utilisation de tel mécanisme de surveillance à granularité fine.

Pour cette expérimentation, nous explorons la capacité à utiliser le modèle de configuration et le support de la reconfiguration à l'exécution pour contourner ce problème. L'expérimentation vise à évaluer la pertinence d'une approche basée sur un système de surveillance reconfigurable optimiste qui ne surveille le système que globalement de manière non intrusive dans des conditions normales et une surveillance fine, localisée et intrusive quand un problème est détecté. Pour ce type d'approche, si il est évident qu'un tel système de surveillance réduit le coût moyen il entraîne aussi un retard pour la découverte des modules au comportement défectueux. L'objectif de l'expérience est de qualifier le gain moyen pour le coût face à la perte de rapidité du diagnostique afin de vérifier si une telle approche peut être bénéfique.

Notre système de surveillance adaptatif optimiste est basé sur les principes suivants :

— **Conception par contrat pour la consommation des ressources**. Chaque composant possède un contrat de qualité de service qui spécifie la consommation de ressources attendue ou précédemment calculée [BJPW99]. Ces contrats spécifient comment un composant utilise la mémoire, les entrées-sorties et le processeur.

— **Activation à la demande et de manière localisée des sondes de surveillances**. Dans des conditions normales notre contrôle du système exécute une surveillance globale du système. Quand un problème est détecté au niveau global, notre système de surveillance active des sondes locales sur des composants spécifiques pour identifier la source du comportement défectueux. Les sondes sont synthétisées à la demande en fonction du contrat de qualité de service du composant afin de limiter leur coût. De cette manière, seules les données nécessaires sont surveillées (par exemple, seule l'utilisation de la mémoire est contrôlée quand un problème de fuite mémoire est détecté).

— **Recherche guidée par une heuristique pour détecter le ou les composants défectueux**. Dans le cadre de cette expérimentation, nous utilisons une heuristique pour réduire le temps nécessaire pour localiser un composant défectueux. Cette heuristique est utilisée pour injecter et activer le contrôle de sondes sur les composants soupçonnés. Il est évident que la latence dans la découverte du ou des composants défectueux est grandement impactée par la précision de cette heuristique. Une heuristique, qui soupçonne en effet uniquement les composants réellement défectueux, réduira cette latence. Dans cette expérimentation, nous proposons d'utiliser les informations contenues dans le modèle de configuration pour bâtir une heuristique efficace.

8.2.1 Contrat de qualité de service

Dans cette expérimentation, nous étendons notre langage de configuration afin d'attacher à chaque composant un contrat de qualité de service lié aux ressources consommées. Ces contrats spécifient la pire consommation du composant en termes de mémoire, processeur, entrées/sorties et de temps pour rendre les services du composant.

Par exemple, pour un composant de type serveur Web simple, nous pouvons définir un contrat sur le nombre d'instructions par seconde qu'il peut exécuter [BH06] et la quantité maximale de mémoire qu'il peut consommer. Le nombre de messages peut être spécifié par composant ou par port du composant comme montré dans le listing 8.1 [1]. Cette extension de contrat suit le principe de séparation entre interface et implémentation [AH01] et permet de détecter si le problème vient de la mise en œuvre du composant ou d'une interaction entre composants. Grâce à ce mécanisme, nous pouvons distinguer entre un composant qui utilise des ressources de manière excessive parce qu'il est défectueux, ou parce que d'autres composants l'appellent de manière excessive.

1. Les exemples de contrats pour une architecture plus complexe peuvent être trouvés ici http://goo.gl/uCZ2Mv.

```
addComponent WsServer650@node0 : WsServer {
  //Specify that this component can use 258 CPU
  //instructions per second,
  cpu_wall_time = '258',
  //Specify that this component can consume a maximum of 15000
  //bytes of memory,
  memory_max_size = '15000',
  //Specify that the contract is guaranteed under the assumption that
  //we do not receive more than 10k messages on the component and
  //10k messages on the port named service
  //(this component has only one port)
  throughput_msg_per_second='all=10000;service=10000'
}
```

Listing 8.1 – Component contract specification example

8.2.2 Un conteneur muni de mécanismes de surveillance dynamique

Scapegoat fournit un nouveu type de nœud qui utilise le modèle de configuration pour guider la recherche de composants défectueux. Dans Kevoree, chaque nœud est en charge de la gestion de l'adaptation. Ainsi, quand il reçoit un nouveau modèle de configuration, le compare sa configuration actuelle avec la configuration exigée par le nouveau modèle de configuration et calcule la liste des actions d'adaptations qu'il doit exécuter. Parmi ces adaptations, le nœud prend en charge le téléchargement des binaires de chaque composant/canal de communication et leurs dépendances, leur chargement en mémoire, l'instanciation de chaque composant/canal de communication et leur assemblage. Pendant ce processus, Scapegoat fait une première vérification statique afin de garantir que le système aura suffisamment de ressources par rapport au contrat de chaque composant avant d'accepter la nouvelle configuration. En parallèle, il instrumente chaque classe des composants afin d'y insérer le code lié aux sondes de consommation de ressources. . Scapegoat utilise les contrats des composants pour vérifier si la nouvelle configuration n'excédera pas la quantité de ressources disponibles sur le système. L'instrumentation du code introduit des sondes pour surveiller la création de chaque objet (pour calculer l'utilisation de mémoire), pour surveiller chaque instruction (pour l'utilisation du processeur) et pour surveiller les appels aux classes qui permettent l'accès aux fonction d'entrées-sorties comme le réseau ou le système de fichiers. Pour instrumenter ces classes, nous utilisons les *Java Agents* qui fournissent la capacité de redéfinir le contenu de classe qui est chargée à l'exécution [BBCP05].

Nous fournissons plusieurs niveaux d'instrumentation qui varient en fonction des informations qu'ils fournissent et en fonction de leur impact sur les performances de l'application :

- La **surveillance globale**(*global monitoring*) n'instrumente aucun composant, il utilise simplement des informations fournies directement par la machine virtuelle Java.
- L'**instrumentation de la mémoire** (*Memory instrumentation* ou *memory accounting*) qui contrôle l'utilisation de la mémoire pour chaque composant.
- L'instrumentation des instructions (*Instruction instrumentation* ou *instruction accounting*), surveille le nombre d'instructions exécutées par chaque service de chaque composant.
- L'instrumentation de la mémoire et des instructions (*Memory and instruction instrumentation*), qui surveille tant l'utilisation de la mémoire que le nombre d'instructions exécutées.

Les sondes sont synthétisées selon les contrats des composants. Par exemple, un composant dont le contrat ne spécifie pas l'utilisation d'entrées-sorties ne sera pas instrumenté pour le contrôle des entrées-sorties. Les sondes peuvent être dynamiquement activées ou désactivées, à l'exception des sondes liées à l'utilisation de mémoire. Les sondes de liées à la surveillance de la

consommation de mémoire doivent rester activées pour garantir que l'on estime correctement toute l'utilisation de la mémoire de la création du composant à la destruction de ce même composant. En effet, la désactivation des sondes de surveillance de la mémoire entraînerait inévitablement la perte d'information sur la création d'objet au sein du composant. À l'inverse, les sondes liées à la consommation processeur ou à la consommation pour les entrées/sorties peuvent être activées à la demande pour vérifier pour la conformité de composant par rapport à son contrat de qualité de service.

Comme nous disposons de sondes pouvant être activées à la demande, afin de minimiser le coûts lié à la surveillance, nous désactivons les sondes et nous les activons uniquement quand un problème est détecté au niveau global. Dans ce cas, une heuristique prédit les composants défectueux les plus probables et nous activons ensuite les sondes de contrôle pertinentes. Cette technique signifie que nous activons uniquement la surveillance à grain fin sur des composants soupçonnés de « mauvaise conduite » (non respect de leur contrat). Après le contrôle de ce sous-ensemble de composants, si un de ces composants est en effet défectueux, le système de surveillance a terminé et décide que ces composants sont la source du problème. Si le sous-ensemble de composants est sain, le système continue sa recherche et commence à contrôler un second sous-ensemble très probablement défectueux. Le mécanisme de surveillance adaptatif mise en œuvre dans ScapeGoat est récapitulé dans le listing 8.2.

```
monitor(C: Set<Component>, heuristic : Set<Component>→Set<Component>)
  init memory probes (c | c ∈ C ∧ c.memory_contract ≠ ∅)
  while container is running
    wait violation in global monitoring
    checked = ∅
    faulty = ∅
    while checked ≠ C ∧ faulty = ∅
      subsetToCheck = heuristic ( C \ checked )
      instrument for adding probes ( subsetToCheck )
      faulty = fine-grain monitoring( subsetToCheck )
      instrument for removing probes ( subsetToCheck )
      checked = checked ∪ subsetToCheck
    if faulty ≠ ∅
      adapt the system (faulty, C)

fine-grain monitoring( C : Set<Component> )
  wait few milliseconds // to obtain good information
  faulty = {c | c ∈ C ∧ c.consumption > c.contract}
  return faulty
```

Listing 8.2 – The main monitoring loop implemented in ScapeGoat

Par conséquent, à n'importe quel moment, les applications sont dans un des modes de surveillance suivants :

- **Aucune surveillance**. Le logiciel n'est exécuté sans aucune sonde de contrôle et sans aucune modification.
- **Surveillance globale**. Seule l'utilisation globale de ressource est contrôlée, comme l'utilisation du processeur et l'utilisation de la mémoire de la Machine Virtuelle.
- **La surveillance pleine**. Tous les composants sont contrôlés pour tous les types d'utilisation de ressource. Ceci est équivalent aux approches actuelles à l'état de l'art.
- **La surveillance localisée**. Seul un sous-ensemble des composants est contrôlé.
- **La surveillance dynamique**. Le système de contrôle change de la surveillance globale à la surveillance pleine ou localisée si un comportement défectueux est détecté.

Pour le reste de cette expérimentation, nous utilisons le terme « *all components* » pour la politique de contrôle adaptative dans laquelle le système change du mode de surveillance globale au mode de surveillance pleine au cas où un comportement défectueux est détecté.

8.2.3 Architecture de Scapegoat

. ScapeGoat est construit utilisant Kevoree. ScapeGoat étend Kevoree en fournissant un nouveau Type de nœud et trois nouveaux types de composants :

— le type de nœud « Nœud Contrôlé ». Ce type de nœud traite l'admission de nouveau composant en stockant des informations sur la quantité de ressources disponible. Avant l'admission, il vérifie l'adéquation des contrats par rapport aux ressources disponibles. De plus, il intercepte le chargement de classes pour préparer l'activation des sondes.

— le Composant de surveillance. Ce type composant vérifie le respect des contrats de composants. Il met en œuvre une variante complexe de l'algorithme du listing 8.2. Il communique avec les autres composants pour identifier des composants potentiellement dangereux.

— le composant *détective*. Ce type de composant est un type de composant abstrait permettant de mettre en œuvre, par l'intermédiaire d'un pattern *strategy* [GHJV94], une heuristique de détection de composants défectueux.

— le composant d'adaptation. Ce type de composant est en charge de déclencher une adaptation quand une violation de contrat est détectée. C'est aussi un composant *spécialisable*, plusieurs stratégies d'adaptation peuvent être mises en œuvre dans Scapegoat, comme la suppression de composants défectueux ou le ralentissement de la communication entre composants quand le défaut est lié à une mauvaise interaction entre deux composants.

8.2.3.1 Stratégie de mise en œuvre :

ScapeGoat vise à minimiser le coût de la surveillance en maintenant le système dans son mode de surveillance globale. Pour réaliser ceci, Scapegoat enlève autant de sondes que possible et active seulement les sondes nécessaires. Ceci exige le changement du *bytecode* des classes des composants, quand le mode de surveillance change. Le *bytecode* est changé composant par composant. Nous utilisons la bibliothèque ASM pour exécuter la manipulation bytecode et un agent Java pour obtenir l'accès à et transformer les classes. La manipulation de bytecode est une pratique courante pour le décompte de ressource et le profilage Java [Bin06, BH06, CvE98].

8.2.3.2 Utilisation du modèle à l'exécution pour la construction d'un framework de monitoring efficace

Comme présenté dans la section 8.2.2, notre approche offre un mécanisme pour activer et désactiver la surveillance localisée à grain fin. Nous utilisons alors une heuristique pour déterminer quels composants sont le plus probablement défectueux. Les composants soupçonnés sont les premiers à être contrôlés finement. *Scapegoat* peut être étendu à l'aide de plusieurs heuristiques différentes, qui peuvent liés à une application particulière ou un domaine applicatif. Dans le cadre de notre expérimentation, nous cherchons à évaluer si le Models@Runtime est bénéfique pour l'implémentation de ce type d'heuristique. Pour ce faire, nous proposons une heuristique qui utilise les informations contenues dans le modèle de configuration à l'exécution afin de déduire au plus vite les composants défectueux. L'heuristique se fonde sur le postulat que que la cause d'un mauvais comportement d'une application est probablement liée aux changements les plus récents de cette application. Ceci peut être aussi être édicté de la manière suivante : Les composants récemment ajoutés ou mis à jour sont plus probablement la source d'un comportement défectueux ; Les composants qui interagissent directement avec des composants récemment ajoutés ou mis à jour sont aussi soupçonnés.

L'algorithme utilisé pour classer les composants est présenté plus en détail dans le listing 8.3. En pratique, nous conservons l'historique du modèle de configuration pour raisonner [HFN+14].

```
ranker() : list<Component>
  visited = ∅
  ranking = {}
  for each model M ∈ History
    N = {c | c was added in M}
    Neighbors = ⋃_{c∈N} c.neighbors
    ranking.add N\visited
    visited = visited ∪ N
    SortedNeighbors = sort (Neighbors \ visited)
    ranking.add SortedNeighbors
    visited = visited ∪ Neighbors
  return ranking

sort (S : Set<Component>) : list<Component>
  r = {}
  if S ≠ ∅
    choose b | b ∈ S ∧ b is newer than any other element in S
    r.add b, sort (S\{b})
  return r
```

Listing 8.3 – The ranking algorithm (uses the model history for ranking).

8.3 Protocole expérimental commun

Dans cette section nous présentons les expérimentations et leur résultat et discutons la facilité d'utilisation de notre approche. Nous nous concentrons sur les questions de recherche suivantes d'évaluer la qualité et l'efficacité de Scapegoat :

- **Quel est l'impact des différents niveaux d'instrumentations sur les performances de l'application ?** Notre approche pose comme dogme un coût important pour la surveillance pleine et un coût faible pour le mode de surveillance globale. Les expériences présentées dans la section 8.4 montrent le coût pour chaque niveau d'instrumentation.
- **Est ce que *ScapeGoat* fournit de meilleurs performances que les solutions à l'état de l'art ?** L'expérience présentée dans la section 8.5 met en évidence les gains de performances de notre approche face à un scénario réaliste.
- **Quel est l'impact lié à l'utilisation d'une heuristique au sein de notre approche de surveillance adaptative ?** L'expérience présentée dans la section 8.6 montre l'impact de la taille des applications, du nombre de composants, et de la qualité des heuristiques utilisées par *Scapegoat* sur l'identification des composants défectueux.

L'efficacité de ScapeGoat est évaluée sur deux dimensions : Le coût moyen du système de surveillance et le délai pour détecter un composant défectueux. Cette expérimentation montre le compromis entre les deux dimensions et démontre que *Scapegoat* fournit une solution pragmatique qui augmente légèrement le délai pour détecter un composant défectueux, mais réduit le coût moyen lié à cette détection.

8.3.1 Cas d'étude

Nous avons construit plusieurs cas d'étude fondés sur une application de gestion de crise pour la sécurité civile [2], construite à base de composants Kevoree. Nous utilisons principalement deux fonctionnalités de cette application de gestion de crise. Le premier est lié au système d'information tactique et au tableau de bord. L'équipement donné à chaque pompier contient un ensemble des capteurs qui fournit des données pour l'emplacement actuel du pompier, son battement de coeur, sa température corporelle, ses mouvements d'accélération, la température

2. http://daum.gforge.inria.fr/wiki.php/Firefighters

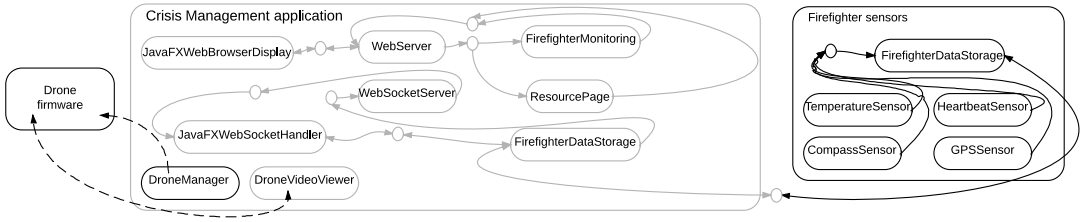


FIGURE 8.1 – Modèle de configuration pour le cas d'étude sur la gestion de crises.

environnementale et la concentration de gaz toxiques. Ces données sont rassemblées et affichées au sein du tableau de bord de gestion de crise, qui fournit une vue globale de la situation. La deuxième fonctionnalité utilise des drones pour capturer la vidéo en temps réel d'un point de vue avantageux.

Le chiffre 8.1 montre l'ensemble des composants qui sont impliqués dans notre cas d'étude : des composants pour gérer : les capteurs des pompiers, les drones et le tableau de bord du système [3]. Dans cette expérimentation, les composants de l'application de gestion de crise sont réels, mais les dispositifs physiques (des drones et des capteurs) sont simulés.

À partir de ce cas d'étude, nous proposons plusieurs extensions : ajout de nouveaux composants ou redondance de composants existants, ajout d'applications externes préalablement encapsulés dans des composants Kevoree (par exemple, Weka, DaCapo), ou modification de composants existants afin d'y injecter des fautes.

8.3.2 Méthodologie de Mesure

Pour obtenir des résultats comparables et reproductibles, nous avons utilisé le même équipement à travers toutes les expériences : un ordinateur portable muni d'un processeur i7-3520M d'Intel cadencé à 2.90GHz, muni de 8GB de mémoire vice et tournant sous une fedora 19 compilé en 32bit. Nous utilisons pour les expérimentations une machine virtuelle 1.7.40 et Kevoree en version 2.0.12. Chaque mesure présentée dans l'expérimentation est la moyenne de dix exécutions différentes dans les mêmes conditions.

L'évaluation de notre approche est fortement dépendante de la qualité des contrats de consommation de ressource attachés à chaque composant. Dans cette expérimentation, nous avons construit ces contrats automatiquement à l'aide de techniques de profilage d'applications classiques. Ainsi, les contrats ont été construits en exécutant plusieurs fois un ensemble de cas de tests sur notre application servant de cas d'étude et sans modifier le modèle de configuration. Ce profilage s'est effectué en deux temps, nous avons exécuté tout d'abord l'application pilote dans un mode de surveillance globale puis nous avons effectué les mêmes exécutions dans un environnent en mode surveillance pleine.

8.4 Expérimentation 11 : Coût lié à l'instrumentation

8.4.1 Protocole expérimental

Notre première expérience compare les différents niveaux d'instrumentation mise en œuvre par Scapegoat pour montrer le coût de chacun. Dans cette expérience, nous comparons les niveaux d'instrumentation suivants : *Aucune surveillance*, *surveillance globale*, *instrumentation de*

3. Plus d'informations sont fournies sur ces composants à l'adresse suivante http://goo.gl/x64wHG

la mémoire, *instrumentation des instructions*, *instrumentation de la mémoire et des instructions* (c'est-à-dire, le mode de surveillance pleine).

Dans cet ensemble d'expériences, nous avons utilisé le benchmark DaCapo de 2006 [BGH+06b]. Nous avons développé un composant Kevoree pour exécuter ce benchmark [4]. Le type de nœud Kevoree introduit par Scapegoat a été configuré pour utiliser le mode de surveillance pleine et les contrats représente les limites supérieures des valeurs de consommations observées [5].

8.4.2 Résultats expérimentaux et analyse

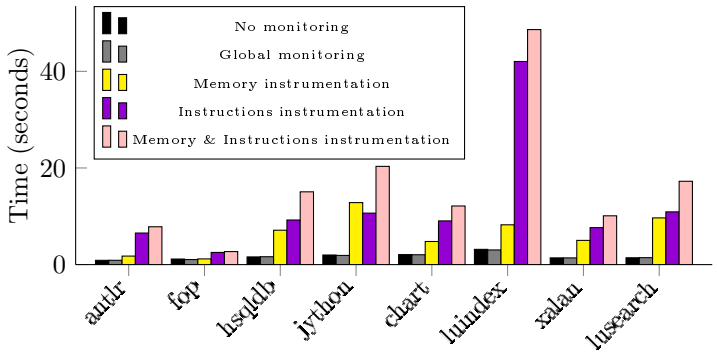


FIGURE 8.2 – Temps d'exécution observé pour les tests utilisant le benchmark DaCapo

Les chiffres présentés dans la figure 8.2 montre le temps d'exécution de plusieurs exécutions du Benchmark DaCapo sous des scénarios différents. Dans un premier temps, nous cherchons à vérifier que le mode de surveillance global n'a en effet aucun impact sur l'application (qu'il n'introduit pas de surcoût face à l'exécution sous le mode « *aucune surveillane* »). Dans un deuxième temps, nous observons que le coût lié à la surveillance fine de la consommation mémoire est inférieure à la surveillance fine de la consommation du processeur. Cette observation est très importante dans notre cas car, comme nous l'avons décrit dans la section 8.2.2, les sondes de surveillance de la mémoire ne peuvent pas être désactivées dynamiquement. Les différents coûts présentés ont été calculés avec la formule suivante :

$$overhead = \frac{WithInstrumentation}{GlobalMonitoring}$$

La moyenne du coût pour la surveillance de la mémoire est de 3.70 tandis que la moyenne du coût pour la surveillance de l'utilisation du processeur est de 6.54. Ces valeurs ne sont pas aussi bonnes que les valeurs rapportées dans [BHMV09] ; la différence est négligeable pour la surveillance de la mémoire mais ils obtiennent une coût inférieur (entre 3.2 et 4.3) pour la surveillance lié au processeur. Cette différence de coût s'explique par le fait qu'ils utilisent un ensemble d'optimisation que nous n'avons pas voulu appliquer. Ces optimisations avaient en effet tendance à augmenter fortement la taille de l'application et par conséquent la consommation mémoire. D'un autre côté, les valeurs que nous observons sont inférieures aux valeurs rapportées dans [BHMV09] pour hprof. Par conséquent, nous considérons que notre solution est comparable aux approches actuelles que l'on trouve dans la littérature.

Les résultats de l'expérimentation, présentés en figure 8.2, montre le coût important lié à la surveillance pleine de l'application. L'approche qui utilise de la surveillance adaptative permet

4. http://goo.gl/V5T6De
5. les scénarios utilisés sont disponible disponibles ici http://goo.gl/FR8LC7.

de réduire fortement le coût moyen de la surveillance. Si ces résultats semblaient évidents, le fait de les garantir au travers de l'expérimentation permet de renforcer l'intérêt de l'approche proposée.

8.5 Expérimentation 12 : Comparaison du coût de la surveillance adaptative face à la surveillance pleine

8.5.1 Protocole expérimental

L'expérience précédente met en évidence l'utilité d'utiliser *la surveillance adaptative*. Cependant, commuter de la surveillance globale à la surveillance pleine ou à la surveillance localisée pour quelques composants amène aussi un coût. Il est en effet nécessaire d'instrumenter le code de composant, il est nécessaire d'activer les sondes. Notre deuxième expérimentation compare justement le coût de la surveillance adaptative face au coût de la surveillance pleine. L'idée est de vérifier que les mécanismes d'adaptation ne sont pas plus coûteux au final que la surveillance elle même.

Le tableau 8.1 présente les tests que nous avons construits pour cette expérience.

Nous avons développé les tests par extension de l'application liée à la sécurité civile. Ces extensions injectent principalement des fautes afin de briser la conformité du contrat de qualité de service (introduction de fuite mémoire, de surconsommation de CPUs, de surconsommation d'entrées/sorties). Les exécutions sont menées de manière répétées ; ainsi un comportement défectueux est exécuté plusieurs fois au cours de la vie de l'application. L'application n'est jamais redémarrée.

Table 8.1 – Caractéristiques des scénarios utilisés.

Nom	Ressource contrôlée	Ressource défaillante	Heuristique	Tâche
UC1	CPU, Memory	CPU	number of failures	Weka, training neural network
UC2	CPU, Memory	CPU	number of failures	dacapo, antlr
UC3	CPU, Memory	CPU	number of failures	dacapo, chart
UC4	CPU	CPU	number of failures	dacapo, xalan
UC5	CPU, Memory	CPU	less number of failures	dacapo, chart
UC6	Memory	CPU	number of failures	Weka, training neural network

8.5.2 Résultats expérimentaux et analyse

La Figure 8.3 montre l'exécution de l'application selon différents scénarios de tests. Chaque scénario utilise une politique de surveillance particulière (*surveillance pleine*, *surveillance adaptative avec activation systématique de toute les sondes pour tous les composants*, *surveillance adaptative avec activation de certaines sondes pour certains composants suspectés*, ou *surveillance globale*). La figure montre que le coût entre le mode *surveillance pleine* et *surveillance adaptative avec activation systématique de toute les sondes pour tous les composants* est clairement impacté par la fréquence de changement entre surveillance globale et surveillance fine. C'est par exemple le cas pour les scénarios UC3 et UC4. en effet, ces scénarios introduisent un composant défectueux qui n'est jamais enlevé ensuite.

L'utilisation de la surveillance adaptative présente une avantage si la somme du coût de la surveillance globale et du coût de la commutation entre le mode de surveillance globale à une surveillance pleine pour tous les composants reste inférieur au coût lié à la surveillance pour

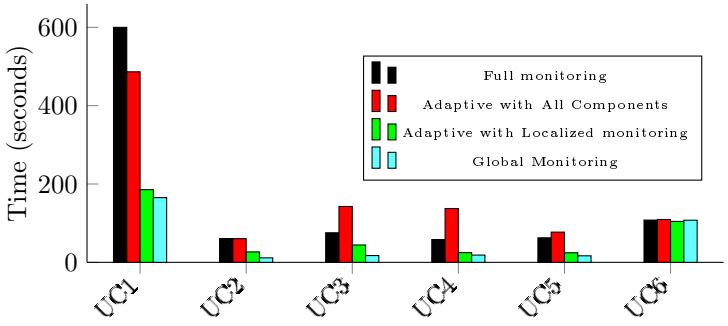


FIGURE 8.3 – Temps d'exécution pour différents scénarios d'exécution et différentes politiques de surveillance.

une exécution donnée. Par conséquent, une fréquence de commutation trop importante diminue l'intérêt d'une telle approche. Le coût de la commutation est clairement induit par le coût d'instrumentation du code des classes pour activer les sondes mais aussi par le remplacement à chaud du code de ces classes. Par conséquent, la surveillance adaptative localisée réduit ce temps de commutation car elle réduit le nombre de classes qui doivent être instrumentées et rechargées. Nous observons ce fait dans le scénarios 3 de la figure 8.3. Dans ce scénario, nous utilisons une heuristique basée sur le nombre de problèmes détectés sur un composant pour le soupçonner. Comme nous exécutons l'application avec un même composant fautif de nombreuses fois, nous obtenons par construction une heuristique presque parfaite pour ce scénario là. Ainsi il est possible de détecter le composant fautif rapidement. Dans ce cas idéal, l'intérêt de la surveillance adaptative localisée permet d'atteindre un gain de 80% par rapport au coût de la surveillance fine activée sur l'ensemble des composants. Nous utilisons l'équation suivante pour calculer ce coût.

$$Gain = 100 - \frac{OurApproach - GlobalMonitoring}{FullMonitoring - GlobalMonitoring} * 100$$

8.6 Expérimentation 13 : Coût lié à la commutation entre modes de surveillance et impact de l'heuristique sur la précision de la localisation de composants défectueux

8.6.1 Protocole expérimental

Comme expliqué dans l'expérimentation précédente, même si nous utilisons un mode de surveillance adaptatif qui permet de commuter entre une surveillance de l'ensemble des composants pour toutes les ressources à un mode de surveillance localisée pour quelques composants et quelques types de ressource, la commutation a un coût. Si ce coût de commutation est trop élevé, le bénéfice de l'approche est perdu. Le but de cette troisième expérience est donc d'observer comment la taille de l'application, le nombre de composants d'une application, la taille des composants (en terme de nombre de classes) impactent l'approche de surveillance adaptative. Dans l'expérience précédente, nous avons aussi observé que la qualité de l'heuristique avait un impact sur la rapidité de détection d'un ensemble de composant défectueux. Nous souhaitons, dans cette troisième expérience, mesurer cet impact.

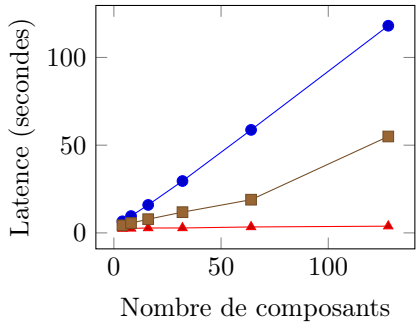


FIGURE 8.4 – Latence pour détecter un composant défectueux pour un composant construit à partir de 115 classes.

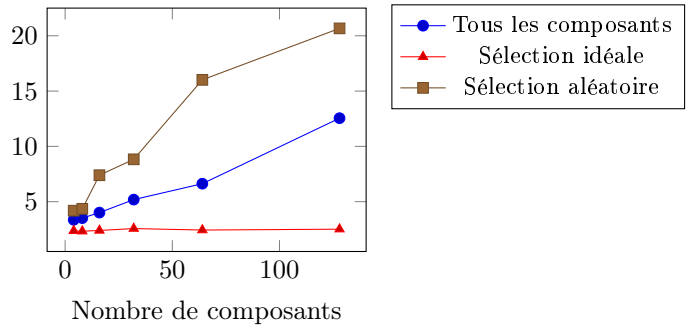


FIGURE 8.5 – Latence pour détecter un composant défectueux pour un composant construit à partir de quatre classes.

Dans cette dernière expérience, nous avons créé deux composants que nous avons introduits dans l'application utilisée précédemment. Ces deux composants ont le même comportement qui consiste à effectuer un test de nombre premier sur des valeurs numériques aléatoirement générées et envoyé ce nombre à un autre composant. Cependant, un des composants entraîne le chargement de 115 classes pour faire ce traitement alors que le deuxième entraîne le chargement que de 4 classes.

Nous utilisons le même scénario élémentaire en faisant varier aussi le nombre d'instances de composants de tests et leur taille. Cela nous permet de faire varier artificiellement la taille d'une application sur deux dimensions : le nombre de composants (complexité de la configuration) et la tailles des composants. Le procédé expérimental a mené à l'exécution de 12 modèles de configuration de tests résultant de la combinaison des contraintes suivantes.

— $N_{comp} = \{4, 8, 16, 32, 64, 128\}$ qui définissent le nombre de composants de l'application.
— $Size_{comp} = \{4, 115\}$ qui définissent le nombre de classe d'un composant.

8.6.2 Résultats expérimentaux et analyse

8.6.2.1 Résultats

Au travers de ces scénarios, nous mesurons le délais pour trouver un composant défectueux et le coût lié à la surveillance. Les figures 8.4 et 8.5 montre la latence pour détecter les composants défectueux par rapport à la taille de l'application. Dans la première figure, les composants sont composés de 115 classes, dans la secondes figures, les composants sont composés de quatre classes.

8.6.2.2 Impact lié à la taille de l'application

Au regard des résultats reportés dans les figures 8.5 et 8.7, nous observons que la taille de l'application a un impact sur la latence pour détecter un composant fautif mais aussi sur le coût de la surveillance. Nous avons aussi calculé le temps nécessaire pour trouver un composant défectueux sous la politique de surveillance pleine après son initialisation (c'est-à-dire le temps nécessaire pour commuter de la surveillance globale à la surveillance pleine). Ce temps est approximativement de deux secondes quelque soit la taille de l'application. C'est pourquoi, commuter d'une surveillance globale à une surveillance fine a un coût si important en terme de performance.

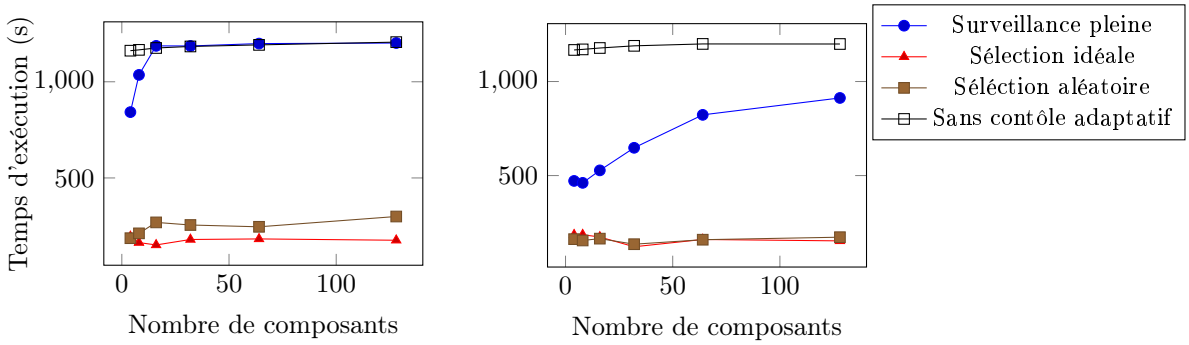


FIGURE 8.6 – Temps d'exécution pour le scénarios muni de composant composé de 115 classes.

FIGURE 8.7 – Temps d'exécution pour le scénarios muni de composant composé de 4 classes.

Ces figures montrent aussi que l'utilisation d'une politique de surveillance dynamique et localisée à la place d'une surveillance pleine permet de réduire l'impact lié à la taille de l'application en réduisant par construction le nombre de composant à instrumenter et à surveiller. Cependant, nous observons aussi que dans ce cas là, l'utilisation d'une heuristique sous-optimale peut avoir un impact important sur le délai pour débusquer le ou les composants fautifs. Ceci s'explique pour les multiples commutations qu'une heuristique aléatoire va engendrer avant de détecter le composant défectueux.

8.6.2.3 Impact lié à la taille des composants

Les figures 8.4 et 8.5 montre que même si nous utilisons une bonne heuristique, la taille des composants impacte le délai nécessaire pour détecter un composant défectueux. Comme la taille de l'application, la taille des composants impacte le coût lié à la commutation entre le mode de surveillance globale et le mode de surveillance localisée. Une bonne heuristique impacte par construction le nombre de ces transitions et donc réduit ce coût.

8.7 Discussions

8.7.1 Limites de validité expérimentale

Nos trois expérimentations 11-12-13 montrent les bénéfices liés à l'utilisation d'un mode de surveillance adaptatif par rapport à des approches plus statiques. Comme pour tout protocole expérimental, l'évaluation de l'approche possède un certain nombre de biais que nous avons essayé d'atténuer.

Biais 1. Toutes nos expériences sont basées autour de la même étude de cas, et il est par conséquent difficile de généraliser à partir d'une unique étude de cas. Cependant nous avons essayé d'atténuer ce biais en utilisant une étude de cas existante représentant un cas d'utilisation réel. Nous avons aussi utilisé des variations dans les modèles de configuration utilisés afin d'évaluer différents points, même si toutes les expériences sont basées sur la même étude de cas. Par conséquent, nos expériences limitent la validité de l'approche aux applications avec les mêmes caractéristiques de l'étude de cas présentée. De nouvelles expériences avec d'autres domaine d'utilisation sont nécessaires pour élargir le périmètre de validation de notre approche.

Par exemple, il serait intéressant de comparer par rapport à l'utilisation d'autres langages s'exécutant sur des machines virtuelles ou aux langages interprétés.

Biais 2. L'évaluation des heuristiques montre principalement le bénéfice obtenu si l'on utilise une heuristique idéale. Notre vision est que la couche de models@runtime permet d'embarquer toute la méta-infirmation, l'historique, les contrats, ...permettant de simplifier la construction de telles heuristiques. Cependant d'autres expériences sont encore nécessaires pour pleinement valider cette conviction.

8.7.2 Travaux en cours

Dans la suite de ces travaux, une deuxième approche est en cours d'expérimentation afin de bénéficier de l'ensemble des informations contenues dans le modèle de configuration pour piloter au mieux les capacités d'isolation et de réservation fournie au niveau système dans les systèmes d'exploitation modernes. Nous introduisons alors un patron/style d'architecture de haut niveau permettant d'isoler de manière efficace un module et en lui réservant une certaine quantité de ressources. Associé à ce patron, nous proposons deux mécanismes pour permettre une instanciation plus efficace de machines virtuelles et une communication efficace entre modules isolés.

Chapitre 9

Kevoree : une synthèse des idées sous la forme d'un projet *open source*

Ce chapitre présente un ensemble d'expérimentations pour l'utilisabilité de l'approche proposée au travers d'une implantation documentée sous la forme d'un projet *open source*. Si François Fouquet est clairement l'architecte en chef sur la partie technique, Kevoree peut-être vu est le résultat de l'intégration des travaux de thèse de Brice Morin, Grégory Nain, Erwan Daubert, François Fouquet, Inti Y. Gonzalez-Herrera et Mohammed Boussa. Cette plate-forme est maintenant aussi très largement utilisée dans plusieurs projets européens dont Heads [1] ou Diversify [2].

Sommaire

9.1 Outillage associé aux modèles de configuration **132**
9.1.1 Notation graphique d'architecture 132
9.1.2 KevScript : DSL de manipulation de modèle d'architecture 133
9.1.3 Model2Code et Code2Model 133
9.1.4 Kevoree IDE, environnement de modélisation d'architecture 138
9.2 Gestion de l'hétérogénéité **139**
9.2.1 Implantation des nœuds 139
9.2.2 Maturité du projet 141
9.3 Dissémination et complexité d'apprentissage liées au langage de configuration **141**

Ce dernier axe de validation cherche à démontrer l'utilisabilité du modèle proposé et la maturité de son prototype au travers de son usage dans différents projets, tel que le projet européen *Heads* en cours ou encore l'usage pour la modélisation et l'adaptation de système de type *Cloud computing*. Le projet Kevoree a donné lieu à la création d'un méta-modèle (plusieurs versions, nous sommes actuellement en version 5) qui spécifie la structure du modèle d'architecture. En reprenant les outils dédiés à la modélisation avant le déploiement de systèmes et les outils des architectures à composants existantes, Kevoree donne lieu à un environnement de développement qui a dû revisiter les outils de modélisation afin de les adapter à un usage lors de l'exécution.

Pour avoir un retour sur le ressenti des développeurs vis-à-vis de l'abstraction proposée, il était nécessaire de construire un prototype qui implante le principe du Model@Runtime mais surtout les outils qui permettent son adoption pour la communauté des développeurs. La suite de cette section détaille les éléments de l'outillage mis en place avant de présenter les projets liés, les informations complémentaires sur cet outillage sont disponibles sur le site du projet [3]. Le projet Kevoree correspond à l'intégration des travaux de Brice Morin, d'Erwan Daubert, de Grégory Nain, d' Inti Y. Gonzalez-Herrera, de François Fouquet et j'espère bientôt de Mohammed Boussa. Cependant il est très important de noter que l'excellente dynamique technique autour du projet est très largement à reporter au crédit de François Fouquet.

9.1 Implantation d'outils et langages dédiés pour la construction de composants et manipulation de modèles de configuration

L'approche Kevoree a été développée à l'aide des outils de l'Ingénierie dirigée par les modèles (IDM). Les modèles structurels proposés ont donné lieu à une implantation d'un méta-modèle suivant le standard EMOF [AK03], [SBMP08]. Les informations liées aux *TypeDefinition* des modèles instances de ces méta-modèles rentrent dans le cadre de la conception continue et font donc l'objet d'un outillage permettant de faire le lien entre le couche de modélisation et la couche de développement. Ces outils sont détaillés en sous-section 9.1.3. Les informations liées aux *Instance* nécessitent des DSL dédiés pour la manipulation et la représentation du modèle d'architecture, ceux-ci sont détaillés en sous-sections 9.1.1 et 9.1.2.

9.1.1 Notation graphique d'architecture

Les notations graphiques sont particulièrement adaptées pour les représentations structurelles car elles permettent de naviguer au milieu des éléments en suivant les relations entre eux, selon les liens de contenance et de liaisons (par exemple lien entre composants). Le modèle Kevoree a donc été enrichi avec un Domain-Specific Language (DSL) graphique qui permet de représenter et manipuler les notions du modèle à composants. Cette notation graphique est illustrée sur un exemple en figure 9.1.

Les composants sont représentés par un rectangle noir avec le nom de l'instance et le nom du type de composant. Les ports requis sont représentés par un rond à droite du composant tandis que les ports fournis sont représentés par un rond à gauche. Les ports de type message ont un fond orange tandis que les ports de type service ont un fond de couleur grise.

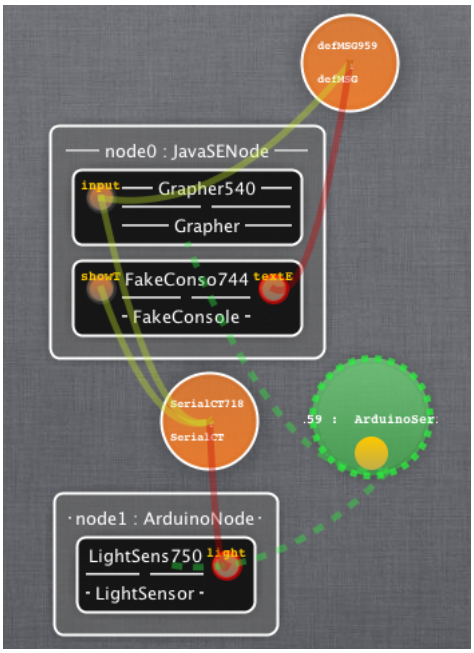
Les Channels sont représentés par un rond orange et leurs relations avec les ports sont représentées par un trait, jaune vers un port fourni, rouge vers un port requis.

Les Groups sont illustrés par un rond vert et la relation avec les nœuds (abonnements) est représentée par un trait en pointillés.

Les nœuds sont représentés par un carré gris et les instances qu'il contient à l'intérieur, le titre de ce carré ayant la syntaxe *titrenœud : titreTypeDefinition*.

3. http ://kevoree.org/

FIGURE 9.1 – Concepts graphiques du modèle Kevoree



9.1.2 KevScript : DSL de manipulation de modèle d'architecture

Pour gérer le passage à l'échelle, la notation graphique doit s'équiper de mécanismes de *slicing* pour limiter la visualisation à un sous-ensemble du modèle. Pour pallier ce manque et automatiser la manipulation de modèle Kevoree, un DSL textuel a été ajouté au projet. A l'inverse de travaux analogues tels FScript de Fractal, KevScript ne manipule que des modèles et n'a aucun lien avec la plate-forme. Le caractère transactionnel de son exécution est alors uniquement lié au processus d'application du modèle qui résulte de la transformation.

Le langage KevScript est embarqué dans les plates-formes, son but est alors de servir de base pour les raisonneurs qui sont caractérisés par de nombreuses opérations de manipulations de modèles. Pour faciliter ces manipulations, le langage KevScript définit des primitives de haut niveau comme par exemple la migration d'un composant d'un nœud à un autre sous la forme : move componentA nodeB. Le listing 9.1 illustre quelques primitives intéressantes du langage KevScript pour illustrer sa syntaxe. Essentiellement le script se divise en plusieurs familles de méthodes qui sont exécutées de façon séquentielle. D'un côté une famille assure la manipulation des instances du modèle à la manière d'un modèle CRUD (create, remove, update, delete) : add, remove.

9.1.3 Model2Code et Code2Model

Definition cyclique et lien avec le modèle de dévelopement : exemple avec le langage Java Le lien avec le modèle de développement est primordial pour permettre une utilisation raisonnable de la conception continue. En d'autres termes il doit être possible d'extraire les informations du modèle de développement pour construire le modèle de conception. A l'inverse, il doit être également possible de générer le code à partir du modèle de conception. En se complétant, ces deux processus permettent d'exploiter le modèle à la fois pour la définition initiale

```
//inclure une definition de modele, depuis un artefact distant ou local

repo "http://dl.bintray.com/andsel/maven/"
repo "https://oss.sonatype.org/service/local/staging/deploy/maven2"
repo "https://oss.sonatype.org/content/groups/public/"
repo "https://oss.sonatype.org/content/repositories/snapshots"
repo "http://repo.maven.apache.org/maven2"

include mvn:org.kevoree.library.java:org.kevoree.library.java.mqttServer:4.0.0
include mvn:org.kevoree.library.java:org.kevoree.library.java.mqtt:4.0.0
...
include mvn:org.kevoree.library.cloud:org.kevoree.library.cloud.api:4.0.0
include mvn:org.kevoree.library.cloud:org.kevoree.library.cloud.docker:4.0.0

add node366, node426 : DockerNode
add node366.node272, node426.node843 : PlatformJavaNode
add group90 : WSGroup
add chan836 : MQTTChannel

attach node366, node426, node272, node843 group90

bind node272.comp992.textEntered chan836
bind node843.comp809.input chan836

set node426.node843.log = 'INFO'
set node426.node843.CPU_CORES = '2'
set group90.port/node426 = '9000'
set group90.port/node843 = '9000'

network node426.ip.lo 127.0.0.1
network node843.ip.lo 127.0.0.1
```

Listing 9.1 – Extrait des commandes KevScript

d'un *ComponentType* par exemple mais également pour sa compréhension et sa modification après un premier développement pour faire évoluer ses capacités en rajoutant par exemple un port. De nombreuses solutions existent pour garantir le lien entre le développement des composants et leur modèle. Pour n'en citer que quelques uns, le projet ArchJava propose d'étendre le langage de développement pour y inclure les primitives de conception des composants, tandis qu'une approche telle que Fraclet propose de décorer de façon non invasive le code Java et de le lire à l'exécution. D'autres approches comme les DSL internes de Scala [4] ou Kotlin [5] permettent également d'extraire des informations qui sont ajoutées au langage de développement sans pour autant le modifier directement comme avec ArchJava.

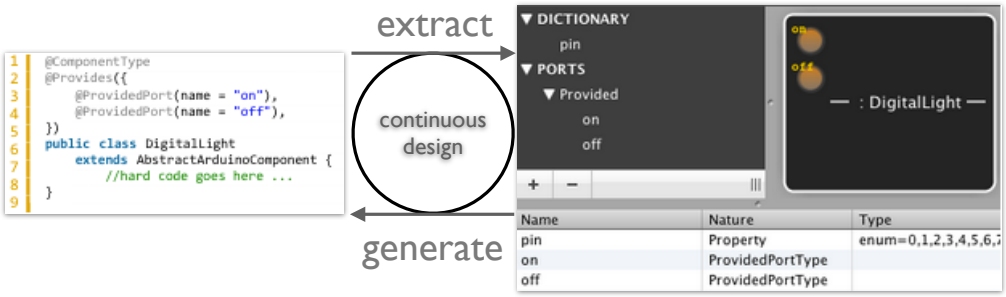
Ce type de méta-information que l'on peut rajouter, tel que les annotations Fraclet, peut être pris en compte au moment de la compilation ou au moment du chargement dans la plate-forme. Dans l'outillage Kevoree le choix s'est porté sur une technique d'annotation non invasive du code, prise en compte directement au moment de la compilation. De ce choix est donc issu un *framework* d'annotations exploitables sur différents langages compatibles sur la machine virtuelle Java, tel que Java, Scala et Kotlin. De plus le fait de prendre les annotations en compte au moment de la compilation permet d'effectuer les optimisations sur le poste du développeur et ainsi permettre le déploiement de Kevoree sur des plates-formes plus modestes telles que Android. Pour le support de langage comme C++ ou Javascript, des approches par convention de nommage ont été utilisées. Le reste de la section suivante détaille ce *framework*, qui permet de décrire les définitions de type de Kevoree sur du code source Java.

Le but de l'outillage de Kevoree est de proposer un cycle de développement continu. D'un côté l'analyse des méta-données telles que des annotations permet d'extraire le modèle d'un

4. http ://www.scala-lang.org/
5. http ://kotlin.jetbrains.org/

code source et de l'autre il est possible de générer ce même code à partir d'un modèle. La boucle illustrée par la figure 9.2 montre l'usage cyclique des ces deux outils. Le générateur de code n'est pas détaillé ici mais permet de générer l'inverse exact du résultat de l'analyseur de code, c'est à dire un squelette de code source Java symétrique à la définition de type du modèle. Cette génération n'est pas destructive, en d'autres termes elle garde l'ensemble du code ajouté par un développeur précédent et met uniquement à jour les éléments manquants vis-à-vis du modèle.

FIGURE 9.2 – Boucle Modèle vers Code et Code vers Modèle



Annotation commune de définition de type L'approche proposée dans Kevoree annotation est fondée sur l'utilisation d'une classe comme point d'entrée pour la définition de type, que ce soit pour un *ComponentType* , un *ChannelType* , un *GroupType* ou un *NodeType* . Il est donc possible de décorer une classe à l'aide d'une annotation pour spécifier le type du point d'entrée, comme illustré dans le listing 9.2.

```
@ComponentType
public class FooKevoreeComponentType extends AbstractComponentType {}
@ChannelTypeFragment
public class FooKevoreeChannelType extends AbstractChannelType {}
@GroupType
public class FooKevoreeGroupType extends AbstractGroupType {}
@NodeType
public class FooKevoreeNodeType extends AbstractNodeType {}
```

Listing 9.2 – Annotation de déclaration de *TypeDefintion*

En supplément il est possible d'hériter d'une classe abstraite ou d'une interface pour pouvoir accéder aux méthodes du *framework* Kevoree . Ainsi il est possible pour un composant d'accéder à ses ports, d'utiliser les primitives de renvoi de messages ou encore d'accéder à la couche Model@Runtime embarquée.

Tous les *TypeDefintion* définissent un cycle de vie, à savoir un état arrêté, démarré, mis à jour (cas de mise à jour de dictionnaire ou de liaisons). Pour la réalisation de ces étapes en Java, il est possible d'annoter une méthode pour chaque étape comme l'illustre le listing 9.3.

La définition du *DictionaireType* commune également à tous les *TypeDefinition* suit également le même principe. Une annotation *@DictionaryType* est ajoutée pour décorer la classe Java. Cette annotation contient un ensemble de *@DictionaryAttribute* qui définissent les attributs du dictionnaire et leurs attributs (énumération, optionnel, etc). Il est à noter ici l'attribut *fragmentDependant* qui signifie que l'attribut prend une valeur différente suivant le nœud d'hébergement, ceci est illustré sur le listing 9.4.

```
public class FooKevoreeType {
  @Start
  public void start(){}
  @Stop
  public void stop(){}
  @Update
  public void update(){}
}
```

Listing 9.3 – Annotation de cycle de vie

```
@DictionaryType({
   @DictionaryAttribute({name="attID",optional=true}),
   @DictionaryAttribute({name="attID2",vals={"val1","val2"},default="attID2"}),
   @DictionaryAttribute({name="attID3",fragmentDependant=true})
})
public class FooKevoreeType { }
```

Listing 9.4 – Annotation de dictionaire type

L'héritage de *TypeDefinition* suivant le modèle Kevoree exploite l'héritage du langage à objets hôte, ici Java. Le fait d'hériter d'une classe ou d'une interface Java définissant déjà une annotation de définition de type suffit pour déclarer la relation d'héritage. Bien évidement l'héritage simple de Java est alors une limite pour le modèle Kevoree , et plusieurs solutions sont disponibles pour déclarer un héritage multiple de composants, par exemple par l'ajout d'autres annotations ou en exploitant la notion de trait des langages tel que Scala ou Kotlin mais ceci n'est pas détaillé ici.

Annotation de définition de *ComponentType* et *Port* La description des contrats de composant s'accompagne en plus de la définition de type d'une définition des ports requis et fournis. Cette définition se fait par annotation de la classe, séparée en deux sections : les ports requis et fournis, comme illustré par le listing 9.5.

Les ports requis nécessaires au fonctionnement du composant sont décrits dans une annotation *@Requires*, *via* des champs *@RequiredPort*. Tous les ports peuvent être de type *Service* ou *Message* , optionnels ou non pour le fonctionnement du composant. Dans le cas d'un port *Service* sa description d'interface peut être définie à l'aide d'une interface Java *via* le champ *className*. Il est à noter ici le champ *needCheckDependency* qui définit si un port doit être utilisé ou non dans les phases de démarrage ou d'arrêt, ce qui a des implications pour la planification du déploiement de ce composant.

La définition des ports fournis suit exactement le même cheminement, le contrat est alors défini dans une annotation *@Provided* et dans des champs *@ProvidedPort*. L'implantation des méthodes des ports est plus complexe. En effet comme nous l'avons publié précédemment [NFM+10], le modèle Kevoree doit intégrer les modèles à services et les communications par évènements. De plus dans le modèle Kevoree rien n'empêche d'avoir plusieurs ports exploitant la même interface message ou service. Ainsi il doit être possible de définir plusieurs ports de service exploitant la même interface Java. Or la plupart des projets liés étudiés dans l'état de l'art exploitent les interfaces Java pour définir les notions de ports de service d'un composant, faisant du même coup l'hypothèse que le composant ne peut définir ces ports plus d'une fois. Pour contourner cette limitation de Java le principe des annotations Kevoree est de séparer la définition des ports fournis en deux parties : d'un côté la définition du contrat, et de l'autre la définition des équivalences entre les méthodes de services et les méthodes Java les implantant.

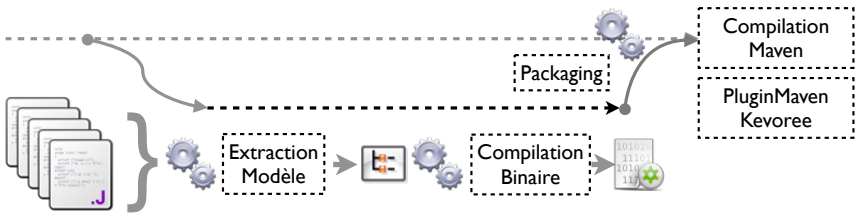
```
@Requires({
        @RequiredPort(name = "out1", type = PortType.MESSAGE, optional = true),
        @RequiredPort(name = "toggle", type = PortType.SERVICE,
    className = ToggleLightService.class, optional = true,
    needCheckDependency = true)
})
@Provides({
    @ProvidedPort(name = "in1", type = PortType.MESSAGE),
    @ProvidedPort(name = "toggle", type = PortType.SERVICE,
    className = ToggleLightService.class)
})
@NonConcurrency("in1","toogle")
@ComponentType
public class FooComponentType extends AbstractComponentType {
  @Port{name="in1"}
  public void reactOnMessage(Object o){}
  @Port{name="toogle",method="toogle"}
  public boolean toogleImpl(){}
}
```

Listing 9.5 – Annotation de dictionaire type

Pour définir ce *mapping* le *framework* définit une annotation *@Port* que l'on peut exploiter pour décorer n'importe quelle méthode Java de la classe du composant type. Ainsi pour le *mapping* d'un port de type message tel que *in1* dans l'exemple du listing une annotation sans paramètre au-dessus d'une méthode prenant un attribut pour récupérer l'objet en transit suffit. Dans le cas d'un port de type service, il faut alors faire correspondre l'ensemble des méthodes du service fourni vers des méthodes Java comme l'illustre le port *toogle*. Kevoree fait ensuite la redirection des appels de service vers les méthodes appropriées, permettant ainsi de définir autant de ports que nécessaire.

Intégration dans les environnements de compilation Le traitement du code source et des annotations des *TypeDefinition* est réalisé au moment de la compilation pour l'intégration avec la langage Java. Ce traitement consiste principalement à produire un fragment de modèle d'architecture ainsi qu'à ajouter des binaires au *packaging* produit par la compilation (JAR en Java). L'intégration avec les outils de compilation largement utilisés par les développeurs de ce langage est donc indispensable pour rendre le développement des *TypeDefinition* viables et permettre la réutilisation de tous les processus industriels d'intégration continue tels que Jenkins [6]. Le projet Apache Maven [7] définit justement un processus de compilation extensible et est largement utilisé pour la plupart des projets Java depuis plus d'une dizaine d'années. Les phases de compilation Kevoree sont donc intégrées dans le processus Maven sous la forme d'un *plugin* dont une illustration du processus est donnée par la figure 9.3.

FIGURE 9.3 – Processus de compilation Kevoree par extension de Maven



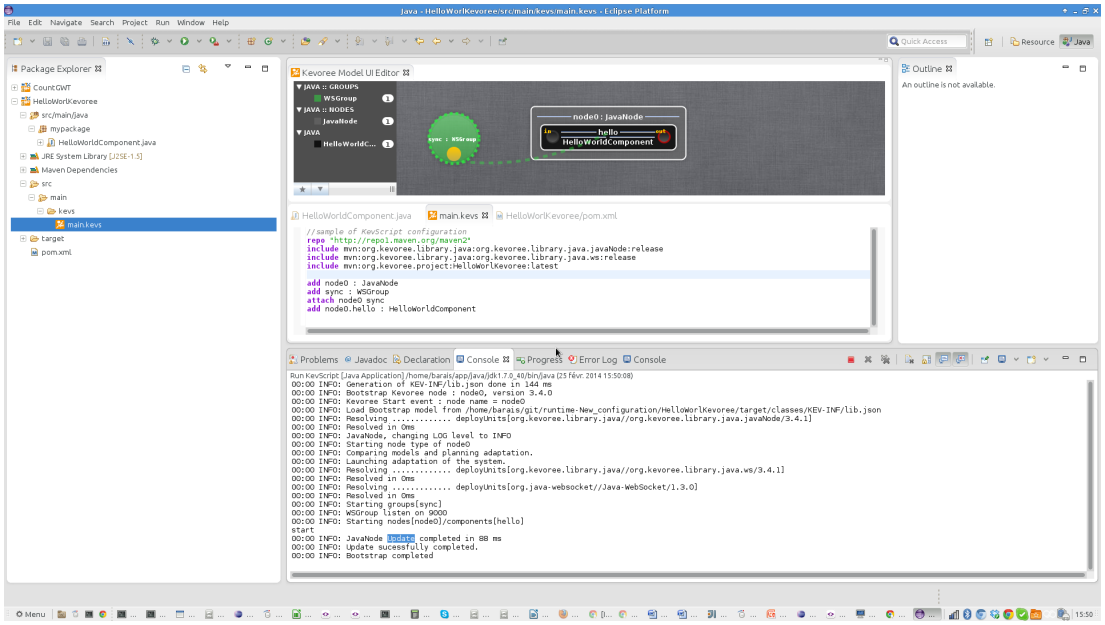
6. http ://jenkins-ci.org/
7. http ://maven.apache.org/

De part cette intégration les fichiers Java annotés sont traités directement pendant la phase de packaging et permet ainsi de manière transparente pour le développeur de rendre compatibles les JAR produits par ses projets pour Kevoree . L'intégration avec l'environnement Maven de Kevoree va bien plus loin que ce simple processus puisqu'elle propose également une lecture des fichiers projets Maven mais également une synchronisation du code utilisateur avec un modèle,etc... La description technique de ces fonctionnements n'est pas détaillée ici mais est néanmoins nécessaire pour l'adoption de l'abstraction par les développeurs Java.

9.1.4 Kevoree IDE, environnement de modélisation d'architecture

L'ensemble des outils de modélisation Kevoree ont été intégrés dans un environnement de développement dédié à la manipulation de modèles Kevoree . Cet environnement illustré par la figure 9.4 est intégré dans des environnements de développement généraliste tels que Eclipse ou IntelliJ. En reprenant le DSL graphique et textuel, il peut lire et modifier à distance les modèles embarqués dans des nœuds de différentes implantations.

FIGURE 9.4 – Environnement de modélisation d'architecture Kevoree intégré à Eclipse



Cet environnent a deux buts : d'un côté le DSL graphique a un aspect pédagogique fort car il permet de réduire la courbe d'apprentissage du modèle à composants, et d'un autre côté il se place en soutien pour la conception des composants lorsqu'il est rapproché d'un environnement de développement généraliste.

En outre, nous proposons un environnement de manipulation de modèle en ligne [8] dont le but est de pouvoir à tout moment éditer la configuration d'un système en cours d'exécution.

8. http://editor.kevoree.org

9.2 Gestion de l'hétérogénéité

L'objectif final de l'approche Kevoree est le pilotage d'un système adaptatif par une couche de modélisation. Mais cette couche de modélisation doit être disponible et exploitable par les éléments déployés sur le système pour rendre la couche de réflexion concrète. Cependant le passage du modèle à la plate-forme ne se fait pas sans des ajustements, car les outils de modélisation n'ont pas été développés pour être déployés directement dans un système. Dans le cadre des thèses de Grégory Nain, Erwan Daubert, ou François Fouquet, ce constat a entraîné des travaux pour adapter ces outils et rendre viable le déploiement d'un modèle dans un système (voir section 3.4). Le but premier du système est la fonctionnalité métier à réaliser, qui ne doit être que peu influencée par l'adaptation dynamique dirigée par le modèle. La technologie de modélisation embarquée doit de ce fait limiter son impact et le *framework Kevoree Modeling Framework*[9] a été développé dans ce sens (voir section 3.4.2.3).

KMF offre une deuxième particularité, comme toute approche générative issu de l'IDM, il permet de fournir une couche homogène qui masque l'hétérogénéité des plate-forme sous jacentes. Cependant, si KMF génère l'ensemble des opérations nécessaires à la manipulation des modèles à l'exécution (diff, merge, load, save, clone, lazyclone, ...) pour différentes plate-formes (C, C++, Java, Javascript, bientôt Go et python), nous avons vue dans la section 3.4 que pour chaque type de nœud, Kevoree demande une spécialisation de la planification et une mise en œuvre des opérations d'adaptation. Le reste de cette sous-section détaille les implantations des différentes plates-formes qui permettent de concrétiser les nœuds d'exécutions sur différents environnements et ainsi gérer à partir d'un unique modèle de configuration un système distribué et hétérogène.

9.2.1 Implantation des nœuds

Le principe même de la conception continue exige que le système soit capable de faire du déploiement à chaud des artefacts logiciels. L'implantation des nœuds type doit donc non seulement fournir l'environnement permettant de faire la comparaison mais aussi de déployer du code. Cette sous-section traite de façon non exhaustive les différents environnements dirigés par Kevoree développés dans ce projet.

NodeType pour les plates-formes Java, OSGi L'implantation de Kevoree étant réalisée en Java, Scala et Kotlin, la plate-forme JavaSE constitue le nœud par défaut dans l'approche. Les artefacts de développement et de déploiement sont alors des fichiers de type JAR contenant des classes compilées prêtes à être intégrées dans le système.

Dans une première version la plate-forme Kevoree a été implantée au-dessus du modèle OSGi afin de pouvoir s'intégrer et piloter des environnements existants déjà construits sur cette plate-forme. Le nœud type Kevoree OSGi ainsi développé permet de prendre comme unité de déploiement les *bundles* OSGi construits par les développeurs auxquels s'ajoutent les méta-informations.

Cependant le non-alignement du modèle Kevoree et le modèle OSGi en terme de modèle de développement a posé de nombreux problèmes de compréhension aux utilisateurs de ces composants. En effet le modèle de développement et les modèles de projet tels que Maven, SBT, Ivy, etc se fondent sur la notion de dépendance qui s'exprime en lien entre JAR. Cette notion de dépendance n'est pas alignée avec la notion de *manifest* OSGi qui exploite une relation entre les packages Java pour exprimer les dépendances.

9. http://kevoree.org/kmf/

Le projet Kevoree ClassLoader (KCL) a été introduit pour simplifier le développement des composants Kevoree et pour permettre une plus grande flexibilité. Ce sous-projet permet de réaliser le chargement à chaud de JAR ainsi que leurs liaisons dynamiques, en prenant un graphe de dépendances. Le nœud standard Kevoree JavaSE exploite donc cette technologie sous-jacente en remplacement de *frameworks* OSGi et permet ainsi un alignement total entre le modèle de développement Java et Kevoree .

NodeType Android, plateforme Dalvik La plate-forme Android exploite une machine virtuelle nommée Dalvik [Ehr10]. Cette dernière exploite un *bytecode* similaire à celui de Java, à ceci près qu'elle exploite une machine à registres là où la JVM exploite une machine à pile. La conversion du *bytecode* Java vers Dalvik est donc possible, et pour l'implantation du nœud Android ceci nous a permis de reprendre les résultats de la plate-forme JavaSE pour la comparaison de modèle. Le chargement à chaud du nouveau *bytecode* est cependant différent : celui-ci a été réalisé comme extension du projet KCL afin que ce dernier puisse charger les fichiers de base Android (.dex) et les lier entre eux suivant un graphe de dépendance. Ainsi les modèles de développement de Java et Android sont alignés grâce au chargement transparent de KCL sur les deux environnements.

NodeType Arduino Le nœud Arduino est lui développé en C et en langage Processing [10]. Son développement est séparé en deux car son processeur ne permet pas de prendre en compte les comparaisons de modèles. Celles-ci sont donc effectuées sur un nœud parent capable de faire tourner l'algorithme de comparaison Java. Le nœud Arduino ne permet pas non plus de faire du chargement à chaud de code, cependant il contient un *framework* d'instanciation dynamique. Pour cela un *framework* spécifique ainsi qu'un ordonnanceur de composant ont été développés afin de permettre l'instanciation de composants et leur planification à chaud.

NodeType C++ Le nœud C++ fournit un environnement pour le développement de composants de channels, ou de groupes en C++. Construits à l'aide de la librairie boost [11], il a nécessité la création d'une API pour le développement de ces éléments inspiré de l'API Java, d'un ensemble de primitive d'adaptation pour les composants C++, d'un processeur de pre-directive de compilation pour pallier le manque d'un système d'annotation en C++. Il a aussi nécessité le développement d'outillage pour les opérations de génération de code depuis le modèle ou de génération de modèle depuis les méta-données présentes dans le code C++. L'implémentation est disponible ici [12].

NodeType Javascript Le nœud JavaScript fournit un environnement pour le développement de composants de *channels*, ou de groupes en JavaScript. Construits principalement par Maxime Tricoire, ingénieur au sein de l'équipe, il a aussi nécessité la création d'une API pour le développement de ces éléments, d'un ensemble de primitive d'adaptation pour les composants C++, d'un pré-processeur pour pallier le manque d'un système d'annotations en JavaScript. Nous avons fait le choix pour la mise en place des méta-données au niveau de code de suivre des conventions de nommage (pour déclarer les propriétés, les ports, ...). Il a aussi nécessité le développement d'outillage pour les opérations de génération de code depuis le modèle ou de génération de modèle depuis les méta-données présentes dans le code JavaScript. Cette implémentation propose en outre un mécanisme de génération de gabarit de composant ou de canaux

10. http ://processing.org/
11. http://www.boost.org/
12. https://github.com/kevoree/kevoree-cpp

de communication à l'aide du projet yeoman [13]. L'implémentation est disponible ici [14]. Deux plate-formes distinctes sont fournis pour les composants construits pour tourner dans le navigateur et ceux les composants construits pour tourner dans un serveur d'applications comme nodeJs [15].

NodeTypes cloud Enfin, les derniers types de nœuds disponibles sont ceux permettant de piloter des infrastructures de machines virtuelles ou des infrastructures de conteneurs systèmes. Nous fournissons pour cela une implémentation pour LXC, Docker ou Jails de BSD comme présenté dans la section 6.1.2. Nous fournissons aussi un support pour AmazonEC2. L'ensemble de ces implémentations est disponibles ici [16].

9.2.2 Maturité du projet

Le projet Kevoree est un projet *open source* dont le développement a commencé début 2010 et est toujours en développement actif grâce à la dynamique de Franǫis Fouquet. Le développement sont principalement distribué maintenant entre l'université du Luxembourg, l'équipe DiverSE à Rennes, et le SINTEF. Au cœur du projet européen Heads, un projet est aussi en cours d'instruction pour une maturation dans le cadre de l'IRT B-COM [17]. Plusieurs tutoriels ont été dispensées dans des conférences et des écoles d'été. Il est utilisé dans plusieurs université pour des cours de spécialité de niveau Master 2.

Le projet Kevoree fournit deux bibliothèques d'artefacts disponibles sur étagère, contenant l'ensemble des composants définis dans les différents cas d'usage. La première est la bibliothèque standard, contenant des nœuds et *channels* types de base nécessaires pour toutes les orchestrations. La deuxième bibliothèque contient des composants plus expérimentaux. La bibliothèque standard est directement accessible comme un magasin d'artefacts depuis l'éditeur de modèle en ligne http://editor.kevoree.org.

9.3 Dissémination et complexité d'apprentissage liées au langage de configuration

L'évaluation d'une abstraction est particulièrement difficile et notamment sa perception par les utilisateurs. Qui plus est dans le cas d'une abstraction pour le développement, la qualité de l'abstraction devrait se mesurer *via* le taux de fautes évitées ou encore le gain en temps de développement. Afin d'améliorer et d'évaluer de manière empirique la qualité des outils fournis, plusieurs travaux pratiques ont été réalisés, avec des étudiants tout d'abord puis avec un panel de chercheurs pour avoir un retour sur la qualité de l'abstraction.

Dans ce cadre, Kevoree est utilisé dans le cadre de travaux pratiques de 4 à 8 heures avec des étudiants de dernière année en Master d'informatique (ISTIC et ESIR, université de Rennes 1). Nous avons aussi donné 4 tutoriels dans des conférences tel que middleware ou comparch, des écoles d'été comme celle du GDR-GPL ou pour des réseaux de formation doctoral européen comme le projet Relate . Enfin, Kevoree a été ou est utilisé dans plusieurs projets européens tels que Nessos, S-Cube, Heads ou Diversify.

Pour le cas des travaux pratiques en présence des étudiants, le but est de faire concevoir un cas d'usage par des développeurs de niveau ingénieur, en mettant en oeuvre plusieurs nœuds de

13. http://yeoman.io/
14. https://github.com/kevoree/kevoree-js
15. http://nodejs.org/
16. https://github.com/kevoree/kevoree-library/tree/master/cloud
17. http://www.b-com.com

calcul et des adaptations dynamiques . L'application de ces expériences a permis de démontrer que la courbe d'apprentissage avec l'abstraction proposée est bonne puisque dans le cadre de travaux pratiques ou des tutoriels, les apprenants réussissent à réaliser la coordination de plusieurs nœuds en moins de 2 h de travail. De même, les premières adaptations mettant en œuvre des reconfigurations structurelles simples (déplacement d'un composant) requièrent moins de 2 h d'apprentissage.

Dans le cadre des tutoriels, nous avons eu systématiquement un panel de 10 à 20 chercheurs. Les tutoriels étaient alors plus ambitieux puisqu'ils proposaient des développements qui allaient de l'assemblage de composants à la réalisation d'un serveur élastique déployé sur une plate-forme *MiniCloud* (simulation d'une infrastructure Cloud en machine virtuelle Java) Les compétences du panel de participants étaient très variables en terme de développement Java et en connaissance des approches à composants. Les compétences sur le système de gestion de projet *Maven* qui a été utilisé dans le tutoriel étaient majoritairement à acquérir par le panel.

Après 2 h de cours puis 2 h de travaux pratiques, la plupart des participants ont réussi à développer et à assembler des composants de manière distribuée. La dernière tâche d'élasticité sur une infrastructure de type *Cloud* nécessite cependant plus de temps. Le langage graphique de modélisation d'architecture a été largement compris par la majorité des participants et semble donc adapté à la représentation du problème. Globalement les participants ont indiqué que la complexité de création d'un nouveau *ComponentType* est légèrement supérieure à la création d'une classe Java et est donc abordable. Enfin, on constate que la création de nouveaux canaux de communication ou de nouveaux types de groupe, pour mettre en œuvre par exemple un algorithme de type Paxos est coûteuse. Cette fonctionnalité de Kevoree est de loin la plus difficile, à cause de la complexité algorithmique de la distribution tout d'abord, mais également pour la gestion de la fragmentation d'un tel groupe (On ne développe qu'un seul module mais celui si sera instancié n fois). Cependant, à l'issue du tutoriel, les participants sont proches de faire ce type d'implantation et en moyenne 10% ont déjà commencé à modifier le code des groupes.

Ces quatre tutoriels donnent un premier niveau de confiance quant à la maturité des outils proposés mais surtout quant à la complexité de prise en main de telles abstractions. En effet la plupart des participants n'avaient qu'une connaissance limitée dans la conception d'architecture à base de composants. Malgré la prise en main des outils annexes tels que Maven, la courbe d'apprentissage courte a permis à la plupart des participants d'arriver à la construction d'un système distribué ouvert et dynamique. Un résultat intéressant est le fait que bien que Kevoree ne cache pas ses communications sous un bus abstrait mais au contraire expose les *ChannelType* et les *GroupType* dans son modèle, les participants ont malgré tout réussi à distribuer leurs composants tout en ayant une bonne confiance dans leurs architectures. L'abstraction ne cachant pas la distribution n'effraie donc pas les utilisateurs, qui au contraire ont apprécié le fait de visualiser les interconnexions entre nœuds. Un bilan chiffré d'une de ces expériences est présente dans la thèse de François Fouquet [Fou13] au regard d'une analyse de retours d'expérience sur un des tutoriels.

Troisième partie

Conclusion et perspectives

Chapitre 10

Bilans d'activités

Ce chapitre conclut ce mémoire de synthèse au travers de plusieurs points : un résumé des contributions scientifiques proposées, un bilan quantitatif en termes de production scientifique, de formation et de coopérations industrielles.

Sommaire

10.1 Bilan scientifique **145**
10.2 Bilan quantitatif **147**
10.2.1 Bilan des publications 147
10.2.2 Bilan en terme de formation 148
10.2.3 Bilan des coopérations industrielles 148

10.1 Bilan scientifique

Dans ce mémoire, nous avons traité sept questions de recherches que nous pouvons classer dans 2 thèmes principaux : l'amélioration des techniques de modélisation en mettant la priorité sur l'hypothèse du monde ouvert, l'utilisation de techniques de modélisation à l'exécution dans le cadre de systèmes adaptatifs, distribués et hétérogènes.

Dans ce premier thème, les contributions ont été principalement autour de problématiques de composition de modèles hétérogènes et de modélisation de la variabilité dans un contexte d'ingénierie système multi-vues. Les résultats de ces travaux ont été intégrés au sein de l'environnement de modélisation Kermeta [1] et du langage CVL, une initiative de standard OMG avortée. L'ensemble des contributions sur la variabilité est maintenant intégré au projet Familiar [2]. Ces travaux sur la modélisation orthogonale de la variabilité dans un environnement multi-vues ont été principalement menés avec Thales dans le cadre des projets MOPCOM-I, MOVIDA, VaryMDE, Gemoc, MeRGE, et bientôt Clarity. Cette collaboration a permis des expérimentations en interne chez Thales des idées et des outils développés. Les trois résultats clés probablement à retenir sur ce thème sont :

1. Les travaux sur la modélisation orthogonale de la variabilité et tout particulièrement, l'expression modulaire de la sémantique de dérivation permettant de spécialiser la sémantique de dérivation en fonction du point de vue de modélisation utilisée et les outils

1. http ://www.kermeta.org
2. http ://familiar-project.github.io/

associés permettant de spécialiser cette sémantique [FFBA+14, VLODBH+14, FFA+13, FBA+13, FBB+12, FBLNJ12, BBLN+12, RCB+13b, RCB+12, RCB+13a, ACC+13, MBJ08, LMV+07, MVL+08, MPL+09, GVM+12].

2. Un travail sur la notion de composition de modèles avec différentes propositions dans le cadre de la modélisation par aspects [ACC+13, GVM+12, MPL+09, MVL+08, MBJ08, CBAK12, MKBJ08, MBJR07, MBB07, KKS+13, BLM+07, CBBJ08, BKB+08, Bar07, CBP+08] généralisé par une définition abstraite de la composition de modèles comme étant une paire correspondance-interprétation. A partir de cette définition, nous avons proposé un cadre théorique qui (1) unifie les représentations des techniques de composition existantes et qui (2) automatise le développement d'outils de composition de modèles [Cla11].

   Ces deux premiers travaux nous ont permis d'adresser RQ2 et RQ3 en proposant des opérateurs de composition de modèles et une gestion de la variabilité afin de migrer un DAS de sa configuration courante vers une nouvelle configuration sans écrire à la main tous les scripts de bas niveau et spécifiques à une plateforme d'exécution donnée [MBJ+09].

3. Une réflexion sur la modularité, le couplage faible et des opérateurs de compositions de modèles supportant l'existant. Sur ce troisième thème, nous avons principalement travaillé à l'intégration de code patrimonial dans des approches MDE [CBJ10] et l'utilisation de la notion de type de modèle introduite par Steel *et al* [SJ07] afin de diminuer le couplage entre les différents artefacts d'un langage (sémantique statique, sémantique opérationnelle, interpréteur, compilateur, éditeurs, ...) et l'arbre de syntaxe abstraite de ce langage [JCB+13, SMM+12, MMBJ09]. Ces deux points ont clairement pour but d'adresser RQ7 :
   — En proposant des opérateurs de composition dans la définition des langages,
   — en intégrant l'exigence du support du code patrimonial dans les chaînes d'ingénierie dirigée par les modèles,
   — en limitant le couplage des artefacts de construction de points de vue par rapport au méta-modèle définissant ce point de vue, nous améliorons le support de la modélisation dans l'hypothèse du monde ouvert en favorisant l'évolutivité des systèmes conçus à l'aide d'une approche de modélisation.

Dans le deuxième thème concernant **l'utilisation de techniques de modélisation à l'exécution**, nous avons tout d'abord démontré le bénéfice lié à l'utilisation de techniques issues de la modélisation à l'exécution pour piloter la reconfiguration de Systèmes dynamiquement adaptables [MNBJ09, MBNJ09, MBJ+09]. Nous avons ensuite travaillé pour adapter l'approche et les abstractions utilisées à la classe des systèmes adaptables distribués et hétérogènes en nous focalisant sur la problématique de la configuration logiciel et du déploiement. Dans ce cadre nous avons, entre autres, démontré la pertinence de l'utilisation du *models@runtime* dans un contexte distribué en réifiant, dans le modèle de configuration, le concept de conteneurs d'exécution pour la gestion du déploiement et le concept de groupe pour la gestion de la couche de réflexivité distribuée [FDP+12b, FDP+12a]. Ces deux travaux nous ont permis d'adresser RQ1 et RQ4, en proposant un langage de configuration commun et une approche de *models@runtime* pour les systèmes adaptatifs, distribués et hétérogènes.

À partir de ces résultats, nous avons démontré l'utilisabilité d'une telle approche pour différents domaines applicatifs :

— l'Internet des objets et le cas particulier de la domotique [NFM+10, NDBJ08, NBFJ09, FMF+12]. Pour ce domaine, nous avons, par l'expérimentation, quantifié le coût (en terme de temps d'adaptation) lié à l'utilisation de techniques de *models@runtime* à l'exécution.

— le Cloud computing [Dau13]. Pour ce domaine, nous avons, par l'expérimentation, vérifié l'applicabilité liée à l'utilisation de techniques de *models@runtime* à l'exécution dans le domaine du cloud en montrant la capacité à manipuler des modèles de grande taille dans des temps raisonnables face au temps de déploiement sur les infrastructures publiques de cloud.
— la surveillance de systèmes partageant un ensemble de ressources limitées [GHBD+14]. Pour ce domaine, nous avons démontré la capacité à mettre en place de outils de surveillance à granularité plastique. Ainsi, la finesse de surveillance s'adapte en fonction du contexte d'exécution.

Ces différentes expérimentations nous ont permis d'adresser RQ5 en démontrant l'applicabilité de l'approche proposée pour différents domaines d'applications et pour différents types d'application.

L'ensemble de ces contributions est matérialisé dans un projet open-source : Kevoree [3] partagé entre le SINTEF en Norvège, le laboratoire SnT au Luxembourg et l'IRISA Rennes et piloté par François Fouquet. Plusieurs tutoriaux ont été ou seront donnés dans des conférences comme Middleware 2013 et 20014 ou Comparch 2014.

En parallèle de ce projet, nous avons identifié un ensemble de lacunes dans les implémentations de *frameworks* de modélisation [FNM+12] et proposé un ensemble de contributions afin de construire un tel *framework* adapté aux problématiques liées à l'utilisation des modèles à l'exécution [HFN+14]. Ces derniers travaux prometteurs permettent d'adresser la question de recherche RQ6 en montrant les limites des solutions actuelles et en proposant un cadre pour une solution permettant d'envisager pleinement l'utilisation de techniques de modélisation à l'exécution.

10.2 Bilan quantitatif

Le bilan de ce projet peut être évalué suivant plusieurs critères : publications, formation, projets industriels.

10.2.1 Bilan des publications

Les publications et les métriques associées à ces dernières sont une manière d'évaluer la qualité d'un travail de recherche même si l'on connaît la limite de ce seul critère [Lan10]. Les travaux de recherches auxquels j'ai participé ont généré un certain nombre de publications dont les deux tableaux ci-dessous donnent un aperçu global. Plus de détails sont donnés dans mon curriculum vitae fourni en annexe de ce mémoire.

RECAPITULATIF	**International(e)**	**National(e)**	**Total**
Revue	7	4	11
Chapitre d'ouvrage	3	0	3
Conférence	44	6	50
Atelier	24	6	30
Rapport de recherche	2	4	6
Total	81	19	100

3. http ://www.kevoree.org

Métriques Google Scholar	Total
Citations	1406
h-index	22
i10-index	39

Parmi ces publications, il est à noter dans les journaux 3 Sosym, 2 STTT et 1 IEEE Computer. Pour les conférences les plus prestigieuses : 2 ICSE (dont 1 track industriel), 3 ASE (dont 1 court), 10 Models, 2 CBSE, 3 SPLC dont (2 courts), 1 GPCE, 2 WICSA (dont 1 court), 1 ICSM, 1 DAIS, 1 NIER@ICSE.

10.2.2 Bilan en terme de formation

En terme de formation, ce projet d'habilitation a été le support de sept thèses de Doctorat soutenues et a permis de former des étudiants de Master. J'encadre ou je co-encadre actuellement six thèses. Sur ces sept jeunes docteurs avec qui j'ai eu la chance de travailler, quatre ont continué sur des carrières académiques sur des postes de chercheurs au SINTEF en Norvège ou au SnT au Luxembourg et un sur un poste d'enseignant chercheur à l'ESEO d'Angers. Trois ont entamé une carrière dans le domaine industriel. Outre l'encadrement d'étudiants, les activités contractuelles m'ont amené à recruter quatre ingénieurs et un postdoctorat sur contrat que j'ai encadré ou co-encadré scientifiquement pour leur activité sur les projets (MOVIDA, Galaxy, DAUM, et Heads).

Type de diplôme	Thèse de Doctorat	DEA/Master M2	Total
Quantité	7 diplômés + 6 en cours	2	15

En complément du tableau suivant qui donne un bilan quantitatif, le détail de mes activités d'encadrement est donné dans mon curriculum vitae fourni en annexe de ce mémoire.

10.2.3 Bilan des coopérations industrielles

Comme mentionné dans la section démarche scientifique en début de ce manuscrit, j'ai eu à cœur d'effectuer ma recherche dans un contexte de collaboration forte avec l'industrie. Cela se traduit par une très forte collaboration avec Thales. Ainsi, suite au projet Movida, nous avons entamé une collaboration bilatérale directe depuis 2011 et nous avons aussi initié trois projets collaboratifs avec eux que ce soit nationalement ou au niveau de l'Europe. Il est à noter que j'ai eu la chance de m'insérer dans un environnement très dynamique. Si je participe à l'ensemble de ces projets, je ne suis ni le coordinateur, ni l'initiateur de l'ensemble de ces projets industriels. J'ai été à l'initiative et le moteur de la collaboration de l'équipe dans ces projets pour huit d'entre eux et j'ai assuré le rôle de responsable/coordinateur pour l'équipe pour dix d'entre eux.

Type de contrat	Européen	National	Bilatéral	Interne	Total
Quantité	5	4	3	2	14

Le bilan des travaux déjà réalisés et des résultats obtenus permet d'envisager de poursuivre les recherches selon trois axes complémentaires. Le fil directeur de mes recherches reste l'adaptation des méthodes d'ingénierie logiciel à l'hypothèse du monde ouvert.

Chapitre 11

Perspectives et Projet de Recherche

Ce chapitre présente plusieurs perspectives de recherche qui sont une suite naturelle aux travaux menés jusque là. Plusieurs d'entres elles sont d'ores et déjà engagées et financées. D'autres ont des visées à plus long terme. Un effort est aussi fait pour montrer l'adéquation et la complémentarité de ce projet de recherche personnel à l'entreprise collective que constitue une équipe projet INRIA. Je conclus ce chapitre par la vision qui motive l'ensemble de ces futurs travaux.

Sommaire

11.1 Projet collectif : Vers une démarche de conception fondée sur la diversité choisie pour améliorer la stabilité et la sécurité du système **150**
11.2 Perspective 1 : Utilisation du models@runtime pour la maîtrise de la diversité **151**
11.2.1 Models@runtime pour la synthèse d'infrastructure de tests de systèmes distribués 152
11.2.2 Models@runtime pour l'ingénierie des langages 152
11.2.3 Models@runtime pour le web 153
11.3 Perspective 2 : Amélioration des approches de modélisation pour le support de la diversité **154**
11.3.1 Opérateurs de composition de modèles adaptés à un ingénieur système pour la gestion explicite de la variabilité 154
11.3.2 Vers un support de la notion de flux de modèles 154
11.4 Perspective 3 : Coopération application/Système pour le support de la diversité **155**
11.5 Vision : Vers un support technique de l'agilité **156**

De nombreuses perspectives de recherches sont encore ouvertes dans la suite de ces travaux. Dans ce cadre, plusieurs projets européens et plusieurs projets nationaux vont permettre de consolider les propositions, transférer les idées, mais aussi évaluer de nouvelles idées présentées dans le projet de recherche suivant.

Ce projet de recherche est pleinement intégré au projet de recherche de l'équipe **DiverSE** qui vient d'être accepté par l'INRIA et sur lequel nous avons travaillé pendant dix-huit mois avec mes collègues sous la direction scientifique de Benoit Baudry. Cette section présente le contexte général de ce projet de recherche autour de la diversité dans le génie logiciel.

La **diversité** apparaît comme une préoccupation essentielle qui couvre toutes les activités de génie logiciel (de la conception à la vérification, en passant par le déploiement et la maintenance) et apparaît dans toutes sortes de domaines, qui reposent sur des systèmes à logiciel prépondérant : de l'ingénierie système aux domaines de l'informatique ambiante combinant les problématiques de l'informatique nuagique, de l'Internet des objets et l'Internet des Services. Si ces domaines semblent apparemment radicalement différents, l'émergence des notions d'agilité, de livraison continue et la prépondérance d'Internet entraînent une convergence des principes qui sous-tendent leur construction et leur validation autour des principes de construction de systèmes adaptables, distribués et hétérogènes. Cette diversité s'observe tout particulièrement dans la diversité des langages utilisés par les acteurs impliqués dans la construction de ces systèmes ; la diversité des environnements d'exécution dans lequel le logiciel doit fonctionner et s'adapter ; la diversité des défaillances contre lesquelles le système doit être capable de réagir. Dans ce projet nous voulons mettre l'accent sur les défis et les opportunités liés à deux niveaux de diversité : celle qui est imposée par le contexte et l'environnement ; celle qui peut être choisie pour permettre l'exploration spontanée et proactive de solutions logicielles alternatives afin d'anticiper et de faire face à des changements imprévus.

11.1 Projet collectif : Vers une démarche de conception fondée sur la diversité choisie pour améliorer la stabilité et la sécurité du système

Cette première partie a pour but de présenter la vision globale dans laquelle nous nous sommes inscrits dans le cadre du projet INRIA DiverSE dirigé par Benoit Baudry. Un article de présentation de ces travaux vient d'être accepté pour IEEE Software en minor revision [ABB+14]

Les notions de réutilisabilité et de modularité sont des éléments clés de l'informatique pour permettre une créativité importante en particulier dans le monde internet. Cependant, la réutilisation a aussi son côté obscur, elle participe à l'émergence d'une monoculture massive. Qu'est-ce que la monoculture en informatique ? Ce concept, souvent utilisé dans le monde des systèmes d'exploitation, fait référence à des infrastructures logicielles massivement dominées par un même socle applicatif [Sta04]. Ainsi le système d'exploitation *Windows* a longtemps été considéré comme une monoculture pour les environnements de bureau. Ce terme de monoculture est issu du domaine agricole où il a été démontré que cultiver la même espèce sans diversité sur des espaces vastes est une mauvaise pratique. De manière similaire, la monoculture logicielle est une mauvaise pratique [Sta04]. Si l'on a souvent cherché à uniformiser l'hétérogénéité dans la lignée d'une approche comme Java et ses slogans "*Write once, run anywhere" (WORA)*, ou parfois "*write once, run everywhere (WORE)*", la monoculture logicielle entraîne souvent un problème de sécurité systémique appelé BOBE "*break once, break everywhere*". En effet, avec une monoculture importante, un attaquant peut exploiter des vulnérabilités à une large échelle. La monoculture dans les systèmes d'exploitation ou dans le domaine des serveurs de bases de données est connu depuis longtemps [Par07], les vulnérabilités récentes sur openssl [Mat14] par exemple rappelle les limites de cette monoculture. Cependant, Internet est en train d'introduire une nouvelle sorte de monoculture que l'on peut appeler la monoculture de *framework* ou la monoculture applicative en référence à l'utilisation de la même application ou du même *framework* applicatif à très large échelle. Prenons deux exemples. Tout d'abord le cas du gestionnaire de contenu *WordPress* [1]. Si l'on considère les 500 000 sites les plus visités [2], nous trouvons 106 412

1. http ://wordpress.org/
2. classement http ://www.alexa.com/

sites utilisant *WordPress*. Parmi ces sites utilisant *WordPress*, 65 558 (64%) ont activé le *plugin Akismet*, qui est vulnérable à l'injection de spam dans les commentaires *WordPress*. 21 849 de ces sites (22,6%) utilisent le *plugin Jetpack*, qui est connu pour ses problèmes de vulnérabilités aux injections SQL. Considérons un deuxième exemple au niveau gestion d'infrastructure, l'émergence de l'utilisation de docker chez Google, Facebook, Amazon, Ebay, Spotify, IBM, ...alors même que docker est encore jeune et que l'on y trouve des vulnérabilités au niveau de l'isolation. Ces deux exemples démontrent trois niveaux de monoculture applicative. Au niveau du système de déploiement, au niveau de l'application (*Wordpress*), au niveau des *plugins* applicatifs. Une seule attaque est donc potentiellement capable de compromettre des milliers de sites Web.

D'un côté, la réutilisabilité et la modularité sont une opportunité pour l'innovation, d'un autre côté, elles ont tendance à faciliter de vulnérabilités systémiques. Ce dernier axe de recherche vise donc à travailler sur la compréhension de cette notion de diversité choisie en permettant de partir de modules logiciels existants et en permettant leur spécialisation automatique de façon à ne garder que les fonctionnalités réellement nécessaires. En combinant ces techniques de transformation automatique afin d'augmenter la diversité dans les applications à l'utilisation de langages de configuration qui pilote ce déploiement, le but est d'obtenir des applications plus robustes, plus fiable et/ou respectant mieux l'anonymat.

Dans le cadre du projet européen FET Diversify dirigé par Benoit Baudry et dans le cadre de la thèse de yeboah-antwi-kwaku qui a débuté en Octobre 2013, nous proposons d'étudier la diversité qui peut être utilisée comme fondement d'un nouveau principe de conception de logiciels. L'augmentation de la diversité dans le système vise à permettre au logiciel de s'adapter à des situations imprévues au moment de la conception. L'approche scientifique du projet diversify et de cette thèse est fondée sur une forte analogie avec les systèmes écologiques, la biodiversité et l'évolution. DIVERSIFY réunit des chercheurs des domaines des systèmes distribués, et de l'écologie afin de traduire les concepts et les processus écologiques dans les principes de conception de logiciels.

11.2 Perspective 1 : Utilisation du models@runtime pour la maîtrise de la diversité

Le deuxième défi, lié à la diversité de logiciels, est qu'il impose d'intégrer le fait que le logiciel doit s'adapter aux changements dans les exigences et de l'environnement. Tout le travail sur Kevoree fournit un environnement complet visant le support de la conception de systèmes adaptatifs distribués et hétérogènes en proposant un langage de configuration homogène. Il offre un support de configuration commun pour les trois niveaux du modèle SPI de l'informatique nuagique. Les modules logiciels ou le modèle de configuration peuvent être développés ou manipulés au travers de différents langages de programmation ou langages de scripts. Conjointement à ce langage de configuration, Kevoree propose i) un *framework* de développement pour la construction d'applications distribués et reconfigurables ; ii) des conteneurs d'applications supportant le déploiement et la reconfiguration de modules applicatifs (Java, Android, C++, Docker, LXC, Jails, ...) ; iii) une librairie standard de modules applicatifs et un ensemble de dépôts pour ces modules.

Kevoree fournit une réponse pour maîtriser la complexité liée à la configuration et à la reconfiguration d'un système distribué et hétérogène. Il fournit une solution concrète pour :

— Construire des applications/services reconfigurables
— Orchestrer de manière cohérente et transactionnelle une reconfiguration transverse au niveau de l'infrastructure, la plate-forme et de l'applicatif.

— Faciliter la création d'applications multi-entité (multi-tenant) en offrant les primitives support à la modélisation de la variabilité
— Unifier la gestion des machines virtuelles (Vmware, VirtualBox, ...), des conteneurs systèmes (docker, lxc, jails, ...) ou des conteneurs applicatifs/serveurs d'applications (servlet, android, osgi, ...).

À partir de cette infrastructure, nous travaillons sur trois utilisations des techniques de modélisation à l'exécution du models@runtime pour améliorer la maîtrise de la diversité.

11.2.1 Models@runtime pour la synthèse d'infrastructure de tests de systèmes distribués

Dans le cadre du projet européen Heads [3], et de la thèse de Mohamed Boussa, qui a démarré en octobre 2013, nous étudions comment l'utilisation du *models@runtime* peut permettre le test de systèmes distribués et hétérogènes à coût raisonnable. L'approche fait le constat que de plus en plus de techniques génératives sont utilisées pour solutionner le problème de l'hétérogénéité, dans l'esprit de la proposition MDA de l'OMG. De telles approches sont aussi pertinentes pour injecter de la diversité, par exemple de la diversité technologique, la même fonctionnalité modélisée une seule fois mais implantée de manière générative dans trois technologies. Face à ces approches, si le test de telles applications reste un enjeu important, le test des générateurs est souvent crucial. Dans ce cadre, la problématique est d'offrir des capacités de vérification d'un certain nombre de propriétés d'un système dynamique distribué et hétérogène à coup faible pour le concepteur de tels systèmes en synthétisant à la demande et de manière maline l'infrastructure, les données et les jeux de tests pour un tel système. L'approche du models@runtime est alors utilisée pour la synthèse de l'infrastructure de tests et du contexte d'exécutions de ces tests.

Ce travail est mené en collaboration avec Gerson Sunye de l'Université de Nantes, Benoit Baudry, Franck Fleurey et Brice Morin du SINTEF.

11.2.2 Models@runtime pour l'ingénierie des langages

La deuxième piste de recherche vise à maîtriser la diversité choisie dans l'ingénierie des langages. En effet, un programme est exécuté conformément à la sémantique du langage de programmation utilisée pour le concevoir. La sémantique fixe généralement le modèle de calcul (MoC) qui détermine le flux d'exécution du programme. Il existe de nombreuses situations où un MoC fixé a tendance à sur-contraindre le flow d'exécution admissible d'un programme : cela contraint la simulation du programme face à différents contextes d'exécution. Cela empêche parfois de trouver le flux d'exécution le plus approprié en fonction de l'environnement, ou encore la diversification de ce flux d'exécution afin de limiter la prévisibilité de calcul (par exemple, pour la cybersécurité).

Travaillant sur la problématique d'ingénierie des langages à l'aide d'une approche de modélisation [JBF11] depuis de nombreuses années, et en s'appuyant sur trois travaux précédents i) la définition modulaire de la sémantique du langage avec un modèle de calcul explicite [CDVL+13] défini dans le cadre du projet Gemoc ; ii) la capacité à adapter le code de source de transformation pour modifier un programme donné [BAM14], et Kevoree pour les mécanismes de bases d'adaptation dynamique, nous explorons la diversification automatique des machines virtuelles (VM) en faisant varier le modèle de calcul à des fins de simulation, à soutenir l'adaptation à l'exécution du flux d'exécution, et de réduire la prévisibilité du calcul de programme. En particulier, nous explorons différentes stratégies pour changer le modèle de calcul, en s'appuyant sur

3. http ://heads-project.eu/

le fait que certaines parties du calcul peuvent être réorganisées de façon aléatoire, remplacées, voire éliminées.

Pour ce faire, nous travaillons à l'identification des zones de calcul plastique dans un code source et dans la sémantique opérationnelle d'un langage grâce à une combinaison d'analyse statique et dynamique. Nous étudions la définition d'une taxonomie de ces zones :

— les zones du code dans lequel l'ordre de calcul peut varier (par exemple, l'ordre dans lequel un bloc d'instructions séquentielles est exécuté) ;
— les zones qui peuvent être enlevées, en gardant les fonctionnalités essentielles [SDMHR11] (par exemple, sauter quelques itérations de la boucle) ;
— les zones qui peuvent être remplacées par des produits alternatifs Code [FL12, SFF+13] (par exemple, remplacer un *try-catch* par une déclaration de retour).

Le but de cette taxonomie et de l'identification de ces zones plastiques dans le code est de les étiqueter comme des zones « *randomizable* » pour la machine virtuelle.

Une fois que nous savons quelles sont les zones dans le code peuvent être « *randomisées* », il est nécessaire de modifier la machine virtuelle pour mettre en œuvre un modèle de calcul qui s'appuie sur la plasticité de calcul. Elle consiste à introduire des points de variation dans l'interpréteur afin de refléter la diversité des calculs admissibles. La modélisation de cette variation admissible permet ensuite d'effectuer ce choix de manière aléatoire ou contraint pour faire face à un ensemble d'exigences particulières.

Ce travail est mené en collaboration avec Benoit Combemale et Benoit Baudry dans le cadre d'une thèse financée sur le projet CLARITY.

11.2.3 Models@runtime pour le web

La troisième piste explorée concerne le domaine du WEB. Dans le cadre d'un projet avec la société *Zenexity*, acteur majeur dans l'ingénierie des applications Web grâce à son cadre de développement logiciel *Play*[4], et au travers de la thèse de Julien Richard Foy nous menons différentes études pour comprendre les bonnes abstractions utilisables dans le domaine du web aussi bien d'un point de vue architecture que d'un point de vue langage de développement. Dans ce cadre, un premier résultat publié à la conférence GPCE 2013 [RFBJ13] a montré comment la définition d'abstractions propres au web pouvait être compilées efficacement aussi bien en cas d'exécution coté serveur ou côté client. Dans les perspectives, nous travaillons afin de trouver les bons mécanismes pour un support efficace des développements d'applications web résilientes. Dans la lignée de cette thèse, nous travaillons sur la problématique de l'adaptation dynamique dans la partie cliente d'une application Web. En effet, on constate que la complexité des applications Web a fortement augmenté ces dernières années et l'on est souvent passé de pages statiques, à l'utilisation de moteurs de *templates* à des *frameworks* riches comme *angularJS*[5] afin de faciliter la réalisation d'applications web monopages. L'avènement d'applications type IDE en mode SaaS (*Software as a Services*) comme cloud9 ou Orion va nécessairement poser le problème du support de l'adaptation dynamique dans les interfaces Web. Si ce challenge nécessite à la fois un modèle de configuration, un support du déploiement à chaud de modules dans le navigateur ou un support de la sécurité pour définir les frontières d'un module, il nécessite aussi un nouveau modèle de configuration prenant en compte à la fois l'assemblage en termes de services fournis et requis mais aussi en terme de composition de Widgets et de structuration du graphe de scène. Un deuxième axe de travail fort consiste donc à offrir un support pour le développement d'applications web monopages reconfigurables en fournissant un langage de (re)configuration du graphe de scène d'une page Web. Les premiers résultats de ce projet sont

4. http ://www.playframework.com/
5. https ://angularjs.org/

disponibles ici [6]. Ici aussi, la bonne maîtrise de ces mécanismes d'adaptation dynamique ouvre la porte à l'injection de diversité choisie pour le domaine du Web.

Ce travail est mené en collaboration avec Maxime Tricoire, Arnaud Blouin, François Fouquet du SnT.

11.3 Perspective 2 : Amélioration des approches de modélisation pour le support de la diversité

11.3.1 Opérateurs de composition de modèles adaptés à un ingénieur système pour la gestion explicite de la variabilité

La gestion de la variabilité est un enjeu important dans une ingénierie systèmes. De nombreux rôles interviennent sur différents points de vue d'un même projet et le modèle de variabilité est souvent transverse à cet ensemble de points de vue. Sur ce point CVL offre une approche pragmatique permettant de partager un même modèle de variabilité en spécifiant au travers d'un modèle de réalisation l'impact de sélection de variants sur les différentes vues du système.

Cependant proposer un langage pour spécifier ce modèle de réalisation reste un défi majeur entre autres quand les futurs utilisateurs de ce langage sont des ingénieurs système et non des informaticiens. En effet, si tout langage de manipulation de modèle *Turing Complete* permet d'envisager la définition de ce modèle de réalisation, la construction d'un langage dédié reste encore un enjeu industriel important. En très forte collaboration avec Thales, cette piste de recherche vise à définir une algèbre construite à partir d'opérateurs de composition de modèles adaptés à un ingénieur système pour la gestion explicite de la variabilité. Les premières expérimentations consistent à intégrer les opérateurs de réalisations de CVL et la notion de pattern de modèle [RBJ07] que l'on trouve maintenant bien outillé de manière industrielle par Thales au sein du consortium eclipse [7] comme opérateur de composition de premier niveau. Ceci permettant de s'approcher de travaux à la TranSAT [BLLMD06] ou SmartAdapter[MBJR07] en combinant une approche de modélisation de la variabilité et des opérateurs de composition issus de la modélisation par aspects.

Un des challenges scientifiques importants ici consiste à vérifier la complétude de ce langage d'un point de vue de son pouvoir d'expression et de valider l'utilisabilité de ce langage par un ensemble d'ingénieurs système ayant l'habitude de modéliser mais pas nécessairement de programmer.

Ce travail est mené en collaboration avec Jérôme Le Noir dans le cadre de différents projets dont le projet européen MeRGE.

11.3.2 Vers un support de la notion de flux de modèles

Si nous observons la tendance/effervescence actuelle dans le domaine du cloud, nous constatons que la notion de container et des approches comme Docker [8] connaisse un vrai succès (près de 600 contributeurs, 10k commits, 2664 Fork, ...). Cette effervescence est dûe entre autres à la facilité de construire des conteneurs réutilisables embarquant une application avec beaucoup moins de lourdeur que les machines virtuelles classiques. L'accroissement du degré de répartition, de la taille des parcs de machines et de services qu'elles hébergent rendent indispensable l'automatisation des opérations de déploiement puis de supervision. En ce moment, la compétition est donc forte en vue de décider quel modèle de configuration, permettant de gérer un

6. https ://github.com/kevoree/kevoree-js
7. https ://eclipse.googlesource.com/diffmerge/org.eclipse.emf.diffmerge.patterns/
8. http ://docker.io

ensemble de conteneurs distribués, va gagner. Entre Kevoree, Occi [9], panamax [10], projectatomic [11], gaudi [12], mesosphere [13], decking [14], kubernetes [15] difficile de dire qui gagnera la bataille même si il ne serait peut-être pas prudent de mises sur Kevoree. Les points positifs résident dans le fait que ces approches proposent presque tous un langage de configuration agnostique technologiquement, généralement basé sur *JSON* ou XML. Comment contribuer suite à notre expérience autour de Kevoree ? Ce qui me semble le plus prometteur consiste à travailler sur le *framework* de modélisation lui-même permettant de construire ces langages de configurations utilisés à l'exécution.

En effet, parmi les caractéristiques d'un tel *framework* de modélisation se trouve sa capacité à prendre en compte l'historique et poser des questions sur des fenêtres de temps glissantes comme proposé dans de nombreux moteurs de traitement des événements complexes (CEP pour *Complex Event Processing*). L'idée est de proposer une définition claire d'un environnement de modélisation supportant à la fois une notion de modèle infini [CTB12, CTB09] et offrant un support efficace pour les opérations classiques effectuées sur les modèles de configuration (synthèse de modèle, clonage, chargement, sauvegarde, exploration d'espace, surveillance sur fenêtre temporisée, etc.). Les travaux autour de KMF [16] [HFN+14, FNM+12] mais aussi d'autres travaux récents de l'équipe [BCB14] ouvre un pan de recherche en modélisation dont l'application au langage de configuration pour la gestion des conteneurs donne un terrain de jeux très prometteur.

Parallèlement à ces travaux, nous explorons aussi, dans le cadre de la thèse de Thomas Degueule, la piste sur les mécanismes de modularisation en phase de méta-modélisation et leur impact sur les performances des opérations classiques effectuées sur les modèles de configuration.

Ce travail est mené en collaboration avec toute une partie de l'équipe du SnT du professeur Yves Le Traon en particulier François Fouquet qui assure la coordination de ces travaux, Benoit Combemale, Arnaud Blouin et Johann Bourcier.

11.4 Perspective 3 : Amélioration de la coopération entre le système d'exploitation et l'espace applicatif pour le support de la diversité

Gérer de la diversité induite comme dans le cas des applications multi-entités (multi-tenant) par exemple, ou introduire volontairement de la diversité implique généralement de déployer des variants d'une même application sur un environnement partageant un ensemble de ressources finies. Si une approche prometteuse consiste à contractualiser par modules logiciels sa consommation de ressources (que ce soit le réseau, la mémoire, les entrées/sorties ou la consommation d'énergie) et surveiller de manière opportuniste le respect de ces contrats [GHBD+14] comme l'approche présentée au chapitre 8, une autre approche consiste à piloter finement les mécanismes de réservation de ressources proposées au niveau système. En effet, Si ce genre de mécanismes est maintenant supporté sur la plupart des systèmes d'exploitation, un des problèmes inhérent est généralement la perte d'information entre l'architecture de l'application ayant une réalité qu'au niveau de l'application en terme de composants, ports, canaux de communication,

9. http ://occi-wg.org/
10. http ://panamax.io/
11. http ://www.projectatomic.io/
12. http ://gaudi.io/
13. https ://mesosphere.io/
14. http ://decking.io/
15. https ://github.com/GoogleCloudPlatform/kubernetes
16. http ://kevoree.org/kmf/

...et la vision système ne voyant que des processus ou des processus légers sans connaissance à priori des potentiels interactions et dépendances entre ces processus. Cette perte d'information empêche généralement de proposer des mécanismes optimaux de gestion de ressources.

Notre vision consiste à offrir un support efficace pour les mécanismes de réservation disponibles au niveau des systèmes d'exploitations, grâce à la conservation à l'exécution des informations architecturales. Dans ce sens, une première expérimentation a permis la mise en place d'un patron architectural et une implémentation pour permettre la réservation de ressources au niveau système.

Ces travaux sont menés sur deux fronts :

— dans le cadre du projet ANR InfraJVM, et de la thèse d'Inti Gonzalez Herrera dont je suis directeur et que je co-encadre avec Johann Bourcier.

— Dans le cadre de la thèse d'Edouard Outin effectué au sein de l'IRT B-COM sous la direction de Jean-Louis Pazat dans laquelle Edouard travaille plus spécifiquement sur les problématiques de consommation d'énergie dans le domaine de l'informatique nuagique distribuée (*multi-datacenters*).

À moyen terme, l'idée, au travers de l'IRT B-COM, est de mettre en œuvre cette vision afin de permettre une gestion efficace des ressources face aux besoins des applications dans un contexte hautement distribué. Dans un tel contexte, la modélisation des réseaux hautement hétérogènes, dynamiques et reconfigurables doit se coupler à une modélisation de la configuration et des besoins des applications. Les mécanismes systèmes d'isolation ou de virtualisation vont nous emmener probablement vers un découplage toujours plus important entre les infrastructures physiques d'exécution et les conteneurs d'exécution logiciels autorisant le déploiement, la configuration à la demande d'applications dans des infrastructures toujours plus virtuelle. En extrapolant autour de ce contexte, il est facilement imaginable que demain, Google annonce un support d'une infrastructure de conteneur pour Android et que Cisco supporte lui aussi de tels mécanismes de conteneurs dans ces infrastructures réseaux dans la lignée de Docker.io, nous nous retrouverons alors avec potentiellement des milliards d'équipements physiques sur lesquels il sera possible de déployer des modules logiciels avec une gestion très fine des ressources utilisées par chacun de ces modules. Dès lors, trouver le bon niveau d'abstraction pour partager et raisonner sur ces informations, coordonner les demandes de déploiement ou de reconfiguration, estimer le montant des ressources requis pour faire tourner les applications restent des verrous scientifiques et un enjeu important.

11.5 Vision : Vers un support technique de l'agilité

Le développement Agile [DD09, Mar03] a un impact énorme sur la façon dont le logiciel est développé dans le monde entier. Nous pouvons voir les méthodes agiles telles que l'*Extrem Programming* (XP) ou les méthodes Scrum méthodes comme une réaction au méthodes de développement traditionnelles, qui mettent l'accent sur une approche rationnelle basée sur une ingénierie résultant de l'intégration d'outils et de processus connu visant une réutilisation maximale. Au contraire, les méthodes agiles relèvent le défi de l'hypothèse du monde ouvert en mettant l'accent sur :

— les individus et leurs interactions plus que les processus et les outils.

— Du logiciel qui fonctionne plus qu'une documentation exhaustive.

— La collaboration avec les clients plus que la négociation contractuelle.

— L'adaptation au changement plus que le suivi d'un plan.

Cependant, il est clair qu'il ne faut pas tomber dans la caricature que l'on trouve parfois dans certains projets agiles où l'on s'interdit de réfléchir et de concevoir pour produire des applications toujours moins maintenables et plus jetables [Mey14]. Au contraire, les fondements

du manifeste agile montre le besoin d'adapter l'infrastructure de développent logiciel se doit de répondre au cycle de vie d'applications de nouvelle génération, agiles et innovantes. Un cadre de développement prenant en compte l'ensemble de ces pré-requis doit donc être proposé aux développeurs et aux administrateurs, pour permettre une gestion efficace de ces outils. Une nouvelle collaboration ou un effacement de la frontière est donc à imaginer entre équipes de développement et de production afin de pouvoir mettre en œuvre, au rythme des besoins business aux cycles de plus en plus courts, les applications de nouvelle génération.

L'infrastructure logicielle de demain devra fournir :

— toujours plus de support du principe de séparations des préoccupations afin faciliter le travail de l'ensemble des individus d'un projet et leurs interactions,
— une frontière toujours plus perméable entre l'espace de conception et l'espace d'exécution pour pouvoir exécuter au plus tôt mais aussi valider et vérifier le logiciel produit,
— toujours plus d'abstraction à la demande et un support des modes de conception multi-vues afin de comprendre une préoccupation et favoriser la collaboration avec le client,
— un support natif de l'anticipation du changement autorisant la composition et la reconfiguration du logiciel à l'exécution mais aussi l'exploration de la diversité des solutions possibles de manières automatiques et transparentes pour le développeur.

Bibliographie

[ABB+14] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, and M. Tricoire. Multi-tier diversification in internet-based software applications. *IEEE Software Computer*, 2014. To appear.

[ACC+13] M. Acher, B. Combemale, P. Collet, O. Barais, P. Lahire, and R. B. France. Composing your compositions of variability models. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, 2013.

[AH01] L. Alfaro and T. Henzinger. Interface theories for component-based design. In T. Henzinger and C. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer Berlin Heidelberg, 2001.

[AK03] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *Software, IEEE*, 20(5) :36–41, 2003.

[Ama] Amazon EC2. http://aws.amazon.com/en/ec2 (last accessed on October, the 16th 2012).

[APW+08] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[BAM14] B. Baudry, S. Allier, and M. Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 149–159, New York, NY, USA, 2014. ACM.

[Bar07] O. Barais. Séparation des préoccupations en phase de méta-modélisation, 2007. http ://www2.lifl.fr/m̃ullera/CoMo07.htm http ://www2.lifl.fr/m̃ullera/CoMo07.htm.

[BBB+98] R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. Architecturing and configuring distributed application with olan. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 241–256, London, UK, UK, 1998. Springer-Verlag.

[BBCP05] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. Jade — a java agent development framework. In R. Bordini, M. Dastani, J. Dix, and A. Fallah Seghrouchni, editors, *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 125–147. Springer US, 2005.

[BBF09] G. Blair, N. Bencomo, and R. France. Models@run.time. *Computer*, 42(10) :22 –27, oct. 2009.

[BBFS08] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *VaMoS'08 : 2nd Int. Workshop on Variability Modeling of Software-intensive Systems*, Essen, Germany, January 2008.

[BBLN+12] O. Barais, B. Baudry, J. Le Noir, et al. Leveraging variability modeling for multi-dimensional model-driven software product lines. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*, pages 5–8. IEEE, 2012.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. How Reuse Influences Productivity in Object-Oriented Systems. *Communications of ACM*, 39(10) :104–116, 1996.

[BC04] E. Baniassad and S. Clarke. Theme : An Approach for Aspect-Oriented Analysis and Design. In *ICSE'04 : 26th International Conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society.

[BCB14] E. Bousse, B. Combemale, and B. Baudry. Scalable Armies of Model Clones through Data Sharing. In *17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Valencia, Spain, 2014. Springer.

[BCL+06] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java : Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36 :1257–1284, September 2006.

[BDPF00] L. Bellissard, N. De Palma, and D. Féliot. The olan architecture definition language. Technical report, C3DS Technical Report, 2000.

[Ben09] N. Bencomo. On the Use of Software Models during Software Execution. In *MISE'09 : Proceedings of the Workshop on Modeling in Software Engineering, at ICSE'09*, 2009.

[BGH+06a] J. Bayer, S. Grard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, and T. Widen. *Software Product Lines*, chapter Consolidated Product Line Variability Modeling, pages 195–242. Number ISBN : 978-3-540-33252-7. Springer Verlag, 2006.

[BGH+06b] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks : Java benchmarking development and analysis. In *OOPSLA '06 : Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BH06] W. Binder and J. Hulaas. Exact and portable profiling for the {JVM} using bytecode instruction counting. *Electronic Notes in Theoretical Computer Science*, 164(3) :45 – 64, 2006. Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006).

[BHB09] C. Ballagny, N. Hameurlain, and F. Barbier. Mocas : A state-based component model for self-adaptation. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO '09. Third IEEE International Conference on*, pages 206–215, Sept 2009.

[BHMV09] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39(1) :47–79, January 2009.

[Bin06] W. Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6) :615–650, May 2006.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7) :38–45, July 1999.

[BKB+08] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke. Composing multi-view aspect models. *Commercial-off-the-Shelf (COTS)-Based Software Systems, International Conference on*, 0 :43–52, 2008.

[BLLMD06] O. Barais, J. Lawall, A. Le Meur, and L. Duchien. Safe integration of new concerns in a software architecture. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 10 pp.–64, March 2006.

[BLM+07] O. Barais, P. Lahire, A. Muller, N. Plouzeau, and G. Vanwormhoudt. Evaluation de l'apport des aspects, des sujets et des vues pour la composition et la réutilisation des modèles. *L'OBJET*, 13(2-3) :177–212, 2007.

[BM76] J. Bondy and U. Murty. *Graph theory with applications.* MacMillan London, 1976.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1) :39–59, February 1984.

[BP89] T. J. Biggerstaff and A. J. Perlis. *Software Reusability Volume I : Concepts and Models*, volume I. ACM Press, Addison-Wesley, Reading, MA, USA, 1989.

[BR02] R. Baldoni and Raynal. Fundamentals of distributed computing. *IEEE Distributed Systems Online*, 3(2) :1–18, 2002.

[BRR05] X. Blanc, F. Ramalho, and J. Robin. Metamodel Reuse with MOF. pages 661–675, 2005.

[BS06] J. Boardman and B. Sauser. System of systems - the meaning of of. *System of Systems Engineering*, 0 :6 pp.–, 2006.

[CBAK12] B. Combemale, O. Barais, O. Alam, and J. Kienzle. Using CVL to Operationalize Product Line Development with Reusable Aspect Models. In *VARY@MoDELS'12 : VARiability for You*, Innsbruck, Autriche, 2012. ACM. VaryMDE (bilateral collaboration between Inria and Thales).

[CBBJ08] F. Chauvel, O. Barais, I. Borne, and J.-M. Jézéquel. Composition of qualitative adaptation policies. In *ASE*, pages 455–458. IEEE, 2008.

[CBJ10] M. Clavreul, O. Barais, and J.-M. Jézéquel. Integrating legacy systems with mde. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 69–78, New York, NY, USA, 2010. ACM.

[CBP+08] F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Expression qualitative de politiques d'adaptation pour fractal. In Y. A. Ameur, editor, *CAL*, volume RNTI-L-2 of *Revue des Nouvelles Technologies de l'Information*, page 119. Cépaduès-Éditions, 2008.

[CBS12] J. J. Cadavid, B. Baudry, and H. A. Sahraoui. Searching the boundaries of a modeling space to test metamodels. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*, pages 131–140. IEEE, 2012.

[CDVL+13] B. Combemale, J. Deantoni, M. Vara Larsen, F. Mallet, O. Barais, B. Baudry, and R. France. Reifying Concurrency for Executable Metamodeling. In M. Erwig, R. F. Paige, and E. Van Wyk, editors, *SLE - 6th International Conference on Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 365–384, Indianapolis, IN, États-Unis, 2013. Springer. CNRS PICS Project MBSAR (http ://gemoc.org/mbsar).

[Cla11] M. Clavreul. *Composition de modèles et de métamodèles : Séparation des correspondances et des interprétations pour unifier les approches de composition existantes*. These, Université Rennes 1, December 2011. final draft.

[CLG+09] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems : A research roadmap. 5525 :1–26, 2009.

[CN01] P. Clements and L. Northrop. *Software product lines : practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CRKGLA89] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *SIGCOMM Comput. Commun. Rev.*, 19 :224–236, August 1989.

[CTB09] B. Combemale, X. Thirioux, and J. Bézivin. On the Need for Infinite Model and a Formal Definition. Technical report, INRIA, July 2009.

[CTB12] B. Combemale, X. Thirioux, and B. Baudry. Formally Defining and Iterating Infinite Models. In R. France, J. Kazmeier, C. Atkinson, and R. Breu, editors, *Proceedings of the 15th international conference on Model driven engineering languages and systems (MODELS'12)*, Lecture Notes in Computer Science, Innsbruck, Austria, October 2012. Springer.

[CvE98] G. Czajkowski and T. von Eicken. JRes : a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

[Dau13] E. Daubert. *Adaptation et cloud computing : un besoin d'abstraction pour une gestion transverse*. These, INSA de Rennes, May 2013.

[DCBM06] A. Duarte, W. Cirne, F. Brasileiro, and P. Machado. Gridunit : software testing on the grid. In *Proceedings of the 28th international conference on Software engineering*, pages 779–782. ACM, 2006.

[DD09] T. Dyba and T. Dingsoyr. What do we know about agile software development ? *Software, IEEE*, 26(5) :6–9, Sept 2009.

[DG08] J. Dean and S. Ghemawat. Mapreduce : Simplified data processing on large clusters. *Commun. ACM*, 51(1) :107–113, January 2008.

[Dij72] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10) :859–866, 1972.

[Ehr10] D. Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[FBA+13] J. B. F. Filho, O. Barais, M. Acher, J. L. Noir, and B. Baudry. Generating counterexamples of model-based software product lines : An exploratory study. In *17th International Conference on Software Product Lines (SPLC'13)*, 2013. Best Student Paper Award.

[FBB+12] J. B. F. Filho, O. Barais, B. Baudry, W. Viana, and R. M. C. Andrade. An approach for semantic enrichment of software product lines. In E. S. de Almeida, C. Schwanninger, and D. Benavides, editors, *SPLC (2)*, pages 188–195. ACM, 2012.

[FBLNJ12] J. a. B. F. Filho, O. Barais, J. Le Noir, and J.-M. Jézéquel. Customizing the common variability language semantics for your domain models. In *Proceedings of the VARiability for You Workshop : Variability Modeling Made Useful for Everyone*, VARY '12, pages 3–8, New York, NY, USA, 2012. ACM.

[FDP+12a] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, A. Blouin, et al. Kevoree : une approche model@ runtime pour les systèmes ubiquitaires. In *UbiMob2012*, 2012.

[FDP+12b] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *DAIS 2012*, Stockholm, Suède, June 2012.

[FF06] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works) http ://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[FFA+13] M. C. Felice, J. B. F. Filho, M. Acher, A. Blouin, and O. Barais. Interactive visualisation of products in online configurators : A case study for variability modelling technologies. In *MAPLE/SCALE 2013 at SPLC 2013 Joint Workshop of MAPLE 2013 – 5th International Workshop on Model-driven Approaches in Software Product Line Engineering and SCALE 2013 – 4th Workshop on Scalable Modeling Techniques for Software Product Lines*, 2013.

[FFBA+14] J. B. Ferreira Filho, O. Barais, M. Acher, J. Le Noir, A. Legay, and B. Baudry. Generating counterexamples of model-based software product lines. *International Journal on Software Tools for Technology Transfer (STTT)*, jul 2014.

[Fid88] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th ACSC*, volume 10, pages 56–66, 1988.

[FK03] I. Foster and C. Kesselman. *The grid 2 : Blueprint for a new computing infrastructure*. Morgan Kaufmann, 2003.

[FL12] S. Forrest and C. LeGoues. Evolutionary software repair. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 1345–1348, New York, NY, USA, 2012. ACM.

[FMF+12] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jézéquel. A dynamic component model for cyber physical systems. In V. Grassi, R. Mirandola, N. Medvidovic, and M. Larsson, editors, *CBSE*, pages 135–144. ACM, 2012.

[FNM+12] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 87–101, Berlin, Heidelberg, 2012. Springer-Verlag.

[FNM+14] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J. Jézéquel. Kevoree modeling framework (KMF) : efficient modeling techniques for runtime use. *CoRR*, abs/1405.6817, 2014.

[Fou13] F. Fouquet. *Kevoree : Model@Runtime pour le développement continu de systèmes adaptatifs distribués hétérogènes*. PhD thesis, Université de Rennes 1, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), 2013.

[Fow04] M. Fowler. Inversion of control containers and the dependency injection pattern, 2004.

[FS04] S. Frénot and D. Stefan. Open-service-platform instrumentation : Jmx management over osgi. In *Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, UbiMob '04, pages 199–202, New York, NY, USA, 2004. ACM.

[FS09] F. Fleurey and A. Solberg. A Domain Specific Modeling Language supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS'09 : ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems*, Denver, Colorado, USA, oct 2009.

[GA99] S. Garfinkel and H. Abelson. *Architects of the information society : 35 years of the Laboratory for Computer Science at MIT*. The MIT Press, 1999.

[GC03] A. Ganek and T. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18, 2003.

[GF99] R. Guerraoui and M. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4) :101–104, 1999.

[GHBD+14] I. Y. Gonzalez-Herrera, J. Bourcier, E. Daubert, W. Rudametkin, O. Barais, F. Fouquet, and J.-M. Jézéquel. Scapegoat : An adaptive monitoring framework for component-based systems. In *WICSA*, pages 67–76. IEEE, 2014.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[GHK+98] T. Graves, M. Harrold, J. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th international conference on Software engineering*, pages 188–197. IEEE Computer Society, 1998.

[Got08] G. Goth. Ultralarge systems : Redefining software engineering ? *IEEE Software*, 25 :91–94, 2008.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[Gue99] R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. *Informatik*, 2 :3–8, 1999.

[Guy] Guy Rosen. Anatomy of an Amazon EC2 Resource ID. http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/ (last accessed on October, the 16th 2012).

[GV08] I. Groher and M. Voelter. Using Aspects to Model Product Line Variability. In *EA@SPLC'08 : 13th International Workshop on Early Aspects at SPLC*, Limerick, Ireland, 2008.

[GVdHT09] J. Georgas, A. Van der Hoek, and R. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10) :52–60, Oct 2009.

[GVM+12] P. Gilles, G. Vanwormhoudt, B. Morin, P. Lahire, O. Barais, and J.-M. Jézéquel. Weaving Variability into Domain Metamodels. *Software & Systems Modeling*, 11(3) :361–383, July 2012.

[HF10] J. Humble and D. Farley. *Continuous delivery : reliable software releases through build, test, and deployment automation.* Pearson Education, 2010.

[HFN+14] T. Hartmann, F. Fouquet, G. Nain, B. Morin, J. Klein, O. Barais, and Y. Le Traon. A native versioning concept to support historized models at runtime. In *MODELS'14 : ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems*, ACM, Sept 2014.

[HGC+06] D. Hughes, P. Greenwood, G. Coulson, G. Blair, F. Pappenberger, P. Smith, and K. Beven. Gridstix : : Supporting flood prediction using embedded hardware and next generation grid middleware. In *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[HHSD10] A. Hubaux, P. Heymans, P.-Y. Schobbens, and D. Deridder. Towards multi-view feature-based configuration. In R. Wieringa and A. Persson, editors, *Requirements Engineering : Foundation for Software Quality*, volume 6182 of *Lecture Notes in Computer Science*, pages 106–112. Springer Berlin Heidelberg, 2010.

[HRPL+95] B. Hayes-Roth, K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. A domain-specific software architecture for adaptive intelligent systems. *IEEE Trans. Softw. Eng.*, 21(4) :288–301, April 1995.

[Hua] Huan Liu. Amazon data center size. http://huanliu.wordpress.com/2012/03/13/amazon-data-center-size/ (last accessed on October, the 16th 2012).

[HW04] G. Hohpe and B. Woolf. *Enterprise integration patterns : Designing, building, and deploying messaging solutions.* Addison-Wesley Professional, 2004.

[IBMa] Google and IBM Announced University Initiative to Address Internet-Scale Computing Challenges. http://www-03.ibm.com/press/us/en/pressrelease/22414.wss (last accessed on October, the 16th 2012).

[IBMb] Google and I.B.M. Join in 'Cloud Computing' Research. http://www.nytimes.com/2007/10/08/technology/08cloud.html?r=1&ei=5088&en=92a8c77c354521ba&ex=1349582400&oref=slogin&partner=rssnyt&emc=rss&pagewanted=print (last accessed on October, the 16th 2012).

[IBMc] IBM Introduces Ready-to-Use Cloud Computing. http://www-03.ibm.com/press/us/en/pressrelease/22613.wss (last accessed on September, the 16th 2014).

[IBRZ00] V. Issarny, L. Bellissard, M. Riveill, and A. Zarras. Component-based programming of distributed applications. In S. Krakowiak and S. Shrivastava, editors, *Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, pages 327–353. Springer Berlin Heidelberg, 2000.

[JB06a] M. Jelasity and O. Babaoglu. T-man : Gossip-based overlay topology management. *Engineering Self-Organising Systems*, pages 1–15, 2006.

[JB06b] S. Jin and A. Bestavros. Small-world characteristics of internet topologies and implications on multicast scaling. *Computer Networks*, 50(5) :648–666, 2006.

[JBF11] J.-M. Jézéquel, O. Barais, and F. Fleurey. Model driven language engineering with kermeta. In *Proceedings of the 3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, GTTSE'09, pages 201–221, Berlin, Heidelberg, 2011. Springer-Verlag.

[JCB+13] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of metalanguages and its implementation in the kermeta language workbench. *Software & Systems Modeling*, pages 1–16, 2013.

[JMB05] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3) :219–252, August 2005.

[JVG+07] M. Jelasity, S. Voulgaris, R. Guerraoui, A. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3) :8, 2007.

[JWEG07] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07 : 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[KAB07] C. Kastner, S. Apel, and D. Batory. A case study implementing features using aspectj. In *SPLC '07 : 11th International Software Product Line Conference*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[Kap01] G. Kapfhammer. Automatically and transparently distributing the execution of regression test suites. In *Proceedings of the 18th International Conference on Testing Computer Software*, 2001.

[KHW03] H. Kreger, W. Harold, and L. Williamson. *Java and JMX : Building Manageable Systems*. Addison-Wesley, Boston, MA, 2003.

[KKR+12] J. Ko, K. Klues, C. Richter, W. Hofer, B. Kusy, M. Bruenig, T. Schmid, Q. Wang, P. Dutta, and A. Terzis. Low power or high performance ? a tradeoff whose time has come (and nearly gone). In G. Picco and W. Heinzelman, editors, *Wireless Sensor Networks*, volume 7158 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2012.

[KKS+13] M. Kramer, J. Klein, J. R. Steel, B. Morin, J. Kienzle, O. Barais, and J.-M. Jézéquel. Achieving Practical Genericity in Model Weaving through Extensibility. In K. Duddy and G. Kappel, editors, *Theory and Practice of Model Transformations - 6th International Conference*, pages 108–124, Budapest, Hongrie, 2013. Springer.

[Kle05] L. Kleinrock. A vision for the Internet. *ST Journal for Research*, 2(1) :4–5, November 2005.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97 : Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[KM85] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *Software Engineering, IEEE Transactions on*, SE-11(4) :424 – 436, april 1985.

[Kos08] R. J. Kosinski. A literature review on reaction time. *Clemson University*, 10, 2008.

[KP02] J. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering,*

2002. ICSE 2002. Proceedings of the 24rd International Conference on, pages 119–129. IEEE, 2002.

[KvS07] A.-M. Kermarrec and M. van Steen. Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5) :2–7, October 2007.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, 1978.

[Lan10] J. Lane. Let's make science metrics more scientific. *Nature*, 464(7288) :488–489, March 2010.

[LMD13] P. Lalanda, J. A. McCann, and A. Diaconescu. *Autonomic Computing - Principles, Design and Implementation.* Undergraduate Topics in Computer Science. Springer, 2013.

[LMV+07] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J.-M. Jézéquel. Introducing variability into Aspect-Oriented Modeling approaches. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, États-Unis, 2007.

[LQ06] P. Lahire and L. Quintian. New Perspective To Improve Reusability in Object-Oriented Languages. *Journal of Object Technology (ETH Zurich)*, 5(1) :117–138, 2006.

[Mar03] R. C. Martin. *Agile software development : principles, patterns, and practices.* Prentice Hall PTR, 2003.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[Mat14] M. B. Matt. La faille heartbleed dans openssl : mettez à jour vos serveurs. 2014.

[MBB07] F. Munoz, O. Barais, and B. Baudry. Vigilant usage of aspects. In *Proceedings of the ADI Workshop at ECOOP 2007*, Berlin, Germany, July 2007.

[MBJ08] B. Morin, O. Barais, and J.-M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, Germany, 2008.

[MBJ+09] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10) :44–51, 2009.

[MBJR07] B. Morin, O. Barais, J.-M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *Models and Aspects workshop, at ECOOP 2007*, Berlin, Germany, Germany, 2007.

[MBNJ09] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132. IEEE, 2009.

[McI68] M. McIlroy. Mass produced software components. In P. Naur and R. B., editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[Mey14] B. Meyer. *Agile ! - The Good, the Hype and the Ugly.* Springer, 2014.

[MFB+08] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *In Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08)*, Toulouse, France, October 2008.

[MFJ05] P. Muller, F. Fleurey, and J. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05 : 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.

[MKBJ08] B. Morin, J. Klein, O. Barais, and J.-M. Jézéquel. A generic weaver for supporting product lines. In *Proceedings of the 13th International Workshop on Early Aspects*, EA '08, pages 11–18, New York, NY, USA, 2008. ACM.

[MMBJ09] N. Moha, V. Mahé, O. Barais, and J.-M. Jézéquel. Generic Model Refactorings. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, États-Unis, 2009.

[MMM95] H. Mili, F. Mili, and A. Mili. Reusing Software : Issues and Research Directions. *IEEE Transactions of Software Engineering*, 21(6) :528–562, 1995.

[MNBJ09] B. Morin, G. Nain, O. Barais, and J.-M. Jézéquel. Leveraging Models From Design-time to Runtime. A Live Demo. In *4th International Workshop on Models@Run.Time (at MODELS'09)*, Denver, Colorado, USA, États-Unis, 2009.

[MPL+09] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.-M. Jézéquel. Weaving Variability into Domain Metamodels. In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, États-Unis, 2009.

[MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECCOP '98 : Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998. *ISBN : 3-540-64737-6.*

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, January 2000.

[MVL+08] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jézéquel. Managing Variability Complexity in Aspect-Oriented Modeling. In *In Proceedings of ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 08)*, Toulouse, France, France, 2008.

[NBF+09] G. Nain, O. Barais, R. Fleurquin, J. Jezequel, et al. Entimid : un middleware aux services de la maison. In *RNTI L 4 CAL 2009 (3e Conference francophone sur les Architectures Logicielles)*, pages 59–72, 2009.

[NBFJ09] G. Nain, O. Barais, R. Fleurquin, and J.-M. Jézéquel. Entimid : un middleware au service de la maison. In O. Zendra and A. Beugnard, editors, *CAL*, volume L-4 of *Revue des Nouvelles Technologies de l'Information*, pages 61–74. Cépaduès-Éditions, 2009.

[NDBJ08] G. Nain, E. Daubert, O. Barais, and J.-M. Jézéquel. Using mde to build a schizophrenic middleware for home/building automation. In P. Mähönen, K. Pohl, and T. Priol, editors, *ServiceWave*, volume 5377 of *Lecture Notes in Computer Science*, pages 49–61. Springer, 2008.

[NFG+06] L. Northrop, P. Feiler, R. Gabriel, J. Goodenough, R. Linger, R. Kazman, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems-the software challenge of the future. *Technical report Software Engineering Institute Carnegie Mellon University ISBN*, 2006.

[NFM+10] G. Nain, F. Fouquet, B. Morin, O. Barais, and J.-M. Jézéquel. Integrating IoT and IoS with a Component-Based approach. In *Procedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, Lille, France, France, 2010.

[OGT+99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[OMG06] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[Opea] OpenShift. https://openshift.redhat.com/app/ (last accessed on September, the 16th 2014).

[Opeb] OpenStack. http://www.openstack.org/ (last accessed on September, the 16th 2014).

[Par07] D. L. Parnas. Which is riskier : Os diversity or os monopoly ? *Commun. ACM*, 50(8) :112–, August 2007.

[PKGJ08] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC'08 : 12th International Software Product Line Conference*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[RBD+09] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 164–182. Springer Berlin / Heidelberg, 2009.

[RBJ07] R. Ramos, O. Barais, and J.-M. Jézéquel. Matching model-snippets. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 2007.

[RCB+12] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Leveraging cvl to manage variability in software process lines. In K. R. P. H. Leung and P. Muenchaisri, editors, *APSEC*, pages 148–157. IEEE, 2012.

[RCB+13a] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Improving reusability in software process lines. In O. Demirörs and O. Türetken, editors, *EUROMICRO-SEAA*, pages 90–93. IEEE, 2013.

[RCB+13b] E. Rouillé, B. Combemale, O. Barais, D. Touzet, and J.-M. Jézéquel. Integrating software process reuse and automation. In *APSEC (1)*, pages 380–387. IEEE, 2013.

[RFBJ13] J. Richard-Foy, O. Barais, and J.-M. Jézéquel. Efficient high-level abstractions for web programming. In J. Järvi and C. Kästner, editors, *GPCE*, pages 53–60. ACM, 2013.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[RUCH01] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10) :929–948, 2001.

[RWLN89] J. Rothenberg, L. E. Widman, K. A. Loparo, and N. R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[SBMP08] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF : Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[Sch06] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), February 2006.

[SDMHR11] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.

[SFF+13] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest. Software mutational robustness. *Genetic Programming and Evolvable Machines*, pages 1–32, 2013.

[SHT06] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams : A survey and a formal semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference*, RE '06, pages 136–145, Washington, DC, USA, 2006. IEEE Computer Society.

[SJ07] J. Steel and J.-M. Jézéquel. On model typing. *Journal of Software and Systems Modeling (SoSyM)*, 6(4) :401–414, December 2007.

[SKRC10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[SMM+12] S. Sen, N. Moha, V. Mah'e, O. Barais, B. Baudry, and J.-M. Jézéquel. Reusable model transformations. *Journal of Software and Systems Modeling (SoSyM)*, 11(1) :111 – 125, 2012.

[SMR+11] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Software : Practice and Experience*, 42(5) :559–583, 2011.

[Sta04] M. Stamp. Risks of monoculture. *Commun. ACM*, 47(3) :120–120, March 2004.

[Sto09] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE '09.*, pages 369 –374, aug. 2009.

[Szy96] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[Ver10] VersionOne. State of agile development, 2010. http ://goo.gl/tpjTpj.

[Vin06] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6) :87–89, November 2006.

[VLODBH+14] D. Van Landuyt, S. Op De Beeck, A. Hovsepyan, S. Michiels, W. Joosen, S. Meynckens, G. De Jong, O. Barais, and M. Acher. Towards managing variability in the safety design of an automotive hall effect sensor. In *18th International Software Product Line Conference (SPLC'14), industrial track*, Florence, Italie, jul 2014.

[Vou08] M. Vouk. Cloud computing–issues, research and implementations. *Journal of Computing and Information Technology*, 16(4) :235–246, 2008.

[WHLA97] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[WJ96] B. Woolf and R. Johnson. The type object pattern. *Pattern Languages of Program Design*, 3, 1996.

[WJ07] J. Whittle and P. Jayaraman. MATA : A Tool for Aspect-Oriented Modeling based on Graph Transformation. In *AOM@MoDELS'07 : 11th Int. Workshop on Aspect-Oriented Modeling*, Nashville USA, Oct 2007.

[ZC06] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

[ZJ06] T. Ziadi and J.-M. Jézéquel. *Software Product Lines*, chapter Product Line Engineering with the UML : Deriving Products, pages 557–586. Number ISBN : 978-3-540-33252-7. Springer Verlag, 2006.

Table des figures

2.1 Openstack architeture from [Opeb] . . . 21
2.2 Illustration du processus Model@Runtime . . . 24

3.1 Vue générale de l'approche . . . 30
3.2 Exemple de cas d'applications . . . 32
3.3 Exemple de modèle CVL appliqué à un modèle de domaine de machine à états finie . . . 34
3.4 Illustration schématique du problème . . . 34
3.5 Vue générale du processus automatisé pour la création de contre-exemples . . . 35
3.6 Vue détaillée du processus automatisé pour la création de contre-exemples . . . 36
3.7 Méta-modèle de machine à états finie utilisé dans le cadre de ces expériences . . . 40
3.8 Dependencies for each new metamodel generated code . . . 41

4.1 Exemple d'application Kevoree . . . 46
4.2 Paradigme Type/Instance, dictionnaire et héritage de type . . . 48
4.3 Cycle de vie d'une instance . . . 49
4.4 Diagramme d'états-transitions montrant l'intégration des ModelListeners dans le processus de Model@Runtime . . . 51
4.5 Méta-modèle Kevoree avec la représentation des primitives d'adaptation . . . 52
4.6 Exemple de configuration nécessitant une planification . . . 53

5.1 Modèle de topologie, expérience #1 . . . 71
5.2 Delai/hop(ms) . . . 72
5.3 Utilisation réseau/nœud(en kbytes) . . . 72
5.4 Topologie pour expérience #2 . . . 73
5.5 Resultats expérience #2 . . . 74
5.6 Topologie pour l'expérience #3 . . . 75
5.7 Réconciliation de modèle émis en concurrence . . . 76

6.1 Machines physiques ayant servi de Cloud . . . 81
6.2 Nombre de lignes de code non générées selon le projet . . . 86
6.3 Ratio de code selon le projet . . . 86
6.4 Processus d'intégration continue . . . 88
6.5 Graphe de dépendances et allocation des composants sur les nœuds . . . 94
6.6 Temps des réceptions des traces . . . 95
6.7 Temps de traitement du modèle . . . 95
6.8 Espace mémoire pour le stockage d'un modèle . . . 96
6.9 Nombre de lignes de code selon le projet . . . 97
6.10 Configuration de base du serveur web . . . 99

7.1 Illustration modèle Kevoree de SmartBuidling 108
7.2 Résultats expérimentaux bruts 109
7.3 Distribution percentile du temps de downtime(en ms) 111
7.4 Résultat expérimental sur la capacité mémoire volatile 113
7.5 Influence de la mémoire persistante sur le temps de démarrage 115

8.1 Modèle de configuration pour le cas d'étude sur la gestion de crises. 124
8.2 Temps d'exécution observé pour les tests utilisant le benchmark DaCapo 125
8.3 Temps d'exécution pour différents scénarios d'exécution et différentes politiques de surveillance. 127
8.4 Latence pour détecter un composant défectueux pour un composant construit à partir de 115 classes. 128
8.5 Latence pour détecter un composant défectueux pour un composant construit à partir de quatre classes. 128
8.6 Temps d'exécution pour le scénarios muni de composant composé de 115 classes. 129
8.7 Temps d'exécution pour le scénarios muni de composant composé de 4 classes. . 129

9.1 Concepts graphiques du modèle Kevoree 133
9.2 Boucle Modèle vers Code et Code vers Modèle 135
9.3 Processus de compilation Kevoree par extension de Maven 137
9.4 Environnement de modélisation d'architecture Kevoree intégré à Eclipse 138

Quatrième partie

Sélection d'articles scientifiques

Taming Dynamically Adaptive Systems Using Models and Aspects*

Brice Morin[1], Olivier Barais[2], Grégory Nain[1] and Jean-Marc Jézéquel[1,2]
[1]INRIA, Centre Rennes - Bretagne Atlantique
[2]IRISA, Université de Rennes1
Campus de Beaulieu
35042 Rennes Cedex - FRANCE
{Brice.Morin | Gregory.Nain}@inria.fr
{Olivier.Barais | Jean-Marc.Jezequel}@irisa.fr

Abstract

Since software systems need to be continuously available under varying conditions, their ability to evolve at runtime is increasingly seen as one key issue. Modern programming frameworks already provide support for dynamic adaptations. However the high-variability of features in Dynamic Adaptive Systems (DAS) introduces an explosion of possible runtime system configurations (often called modes) and mode transitions. Designing these configurations and their transitions is tedious and error-prone, making the system feature evolution difficult. While Aspect-Oriented Modeling (AOM) was introduced to improve the modularity of software, this paper presents how an AOM approach can be used to tame the combinatorial explosion of DAS modes. Using AOM techniques, we derive a wide range of modes by weaving aspects into an explicit model reflecting the runtime system. We use these generated modes to automatically adapt the system. We validate our approach on an adaptive middleware for home-automation currently deployed in Rennes metropolis.

1 Introduction

Society's increasing dependence on software-intensive systems is driving the need for dependable, robust, continuously available adaptive systems. Such systems often propose many variability dimensions with many possible variants, leading to a wide number of possible configurations that is difficult to integrally check at design-time because of time and resource constraints. For example, associations and public institutions of the metropolis of Rennes are working together on a project which aims at allowing dependent people to stay at home as long as possible. Due to the large scale of the project, and the diversity of disabilities that have to be considered, the deployment context will be different for each equipped house. Furthermore each deployment context is going to continuously evolve along with the evolution of the person's disabilities.

This ability to evolve a system at runtime is one critical aspect of achieving continuously availability. Many popular programming frameworks such as OSGI [33] or Fractal [7] now provide support for dynamic adaptation through extension mechanism such as plugins or variability mechanism through introspection and reconfiguration API. However the high variability of crosscutting and non-crosscutting features in adaptive systems introduces an explosion of possible runtime system configurations (often called modes). When new features are introduced at deployment time (vs. design time), we also have to make sure that they do not lead the system into unwanted modes. Besides, due to the fact that features are often partially independent, the (implicit) state-machine representing the path between modes is highly connected, leading to a nearly quadratic explosion of transitions between modes [5, 35]. The inefficacy of the variability and extension mechanisms to tame the high-number of modes transitions might lead to several undesirable consequences related to Dynamically Adaptive System maintainability, including partial duplication of reconfiguration scripts or the non-cover of all the modes transition, etc.

Aspect-Oriented Modeling (AOM) was initially introduced to improve the modularity of software [11, 21], complementary to Model Driven Engineering (MDE)

*This work was partially funded by the DiVA FP7 European project (STREP) (See http://www.ict-diva.eu/) and the S-Cube European Network Of Excellence on Software Services and Systems (See http://www.s-cube-network.eu/).

to link models to the real world [14]. In [24], we have proposed a first approach that leverages AOM and MDE to manage variability at runtime. It relies on the notion of aspect models, that can be woven into an explicit model of the runtime configuration seating on top of the running system. Actual mode switches between runtime configurations are then triggered by reconfiguration scripts automatically generated based on the differences between the initial model and the newly woven one. The result of this approach is that modes becomes somehow implicit from the point of view of the system designer, and new modes can appear when new aspects are introduced during the life of the system. It is thus no longer possible to statically validate every accessible mode and each mode transition.

In this paper, we show how aspects can help designers determine interactions between dynamic variants and how runtime models can be used to validate new configurations on the fly, before committing them on the running system (making it easy to roll-back when a configuration is not valid). Indeed, once the system has been deployed, new variability dimensions and variants that have not been foreseen may appear while the system is running and cannot be stopped. In this case, it is very useful to validate configurations on the fly before actually adapting the running system.

The reminder of this paper is organized as follows. Section 2 describes the general process for managing dynamic variability. Section 3 presents how we leverage AOM and MDE techniques and tools to determine interactions and validate configurations on the fly. Section 4 outlines how our approach was validated in a home-automation context deployed in the metropolis of Rennes. Section 5 discusses related work and section 6 concludes by presenting a set of open research problems based on our experience.

2 Process Overview

This Section presents our approach for managing variability at runtime [24]. The overall process is described in Figure 1.

2.1 Maintaining a High-Level Representation of the Running System

Maintaining a model at runtime [4] representing the running system (Figure 1, step 1) allows us to reason on the model and manipulate the model independently from the running system. Using high-level abstractions, we can discard all the platform-specific runtime information we do not need and reason more easily and more efficiently in the next steps of the process.

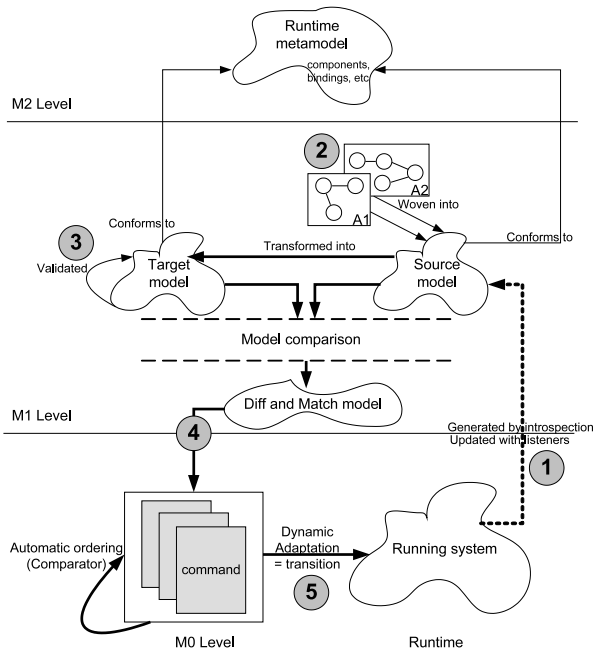


Figure 1. MDE and AOM for Dynamic Adaptation

Recent component-based middleware platforms like OSGi [33] propose introspection APIs that allows discovering the architecture of a running system. We use these APIs to collect and format relevant information in the form of a platform-independent and high-level model. In the case of large systems, using introspection in order to generate a reference model from scratch may be time-consuming especially when only small changes appear. To tackle this issue, we observe the architectural reconfigurations appearing in the running system in order to update the model. This limits the flow of data manipulated in the system. Moreover, recent middleware platforms already propose this kind of observers or propose mechanisms to easily implement such kind of observers [6, 7]. Note that the changes that may affect the source model are not directly reflected to the running system.

2.2 On-demand Construction of Configurations with Aspects

In order to manage variability and avoid the combinatorial explosion of artifacts needed to support this variability, we propose to focus on variation points and variants instead of focusing on whole configurations. A variability dimension is a particular concern that may be realized in different ways. We use as-

pects to represent the different variants of a variation point. Using Aspect-Oriented weavers, whole configurations can be built on-demand by selecting a set of aspects as illustrated in Figure 1, step 2. In practice, we use SmartAdapters [18, 25] an Aspect-Oriented Modeling (AOM) tool for weaving aspects at a model level. However, the approach presented in this paper is not dependent from SmartAdapters and other AOM tools like MATA [13] could also be used. Components present in all the configurations constitute a base model where aspects are woven.

SmartAdapters has formerly been applied to Java programs and UML class diagrams [18]. More recently, we have generalized this approach to any domain-specific modeling language [23]. This allows us to leverage the notion of aspect for runtime models representing at a high level of abstraction the architecture of a system at runtime. SmartAdapters automatically generates an extensible Aspect-Oriented Modeling framework specific to our metamodel.

In SmartAdapters, an aspect is composed of three parts: *i)* an advice model, representing what we want to weave, *ii)* a pointcut model, representing where we want to weave the aspect and *iii)* weaving directives specifying how to weave the advice model at the join points matching the pointcut model. The advice model is a model fragment representing a given concern. In our case, it represents a pre-assemby of components that may not be fully specified. The pointcut model is also a model fragment that is parameterized by roles (See [31]), equivalent to wildcards in AspectJ pointcuts [15, 16]. Both the advice model and the pointcut model are defined using the concrete syntax of the domain. Finally, weaving directives specify how to integrate the advice model into the target model, using a generated domain-specific action language [23].

We (optionally) extend each aspect with a context describing when to trigger the weaving of aspects. A context is a slice of the environment describing when the aspect is useful and its impact on QoS properties. For example, a buffering aspect can *optimize* the bandwidth of the network if the system has enough free memory (*e.g.* $>$ 512 Mb) and if the bandwidth is saturated (*e.g.* $>$ 90%). Aspects with a context are chosen according to the execution context and the QoS properties to optimize [12]. Aspects with no context can be manually triggered by the user.

Figure 2 illustrates an internalization (`I18N` for short) aspect. The pilot of the case study is currently deployed in Rennes. Rennes is an international city hosting many students from different countries where lots of different languages are spoken. Within a single day, people from different countries may transit in the home. Systematically translating all the information in all the possible languages may cause an information overhead that could make information difficult to catch. This is why internationalization should be handled dynamically.

In order to design this aspect, we leverage the ability of SmartAdapters to integrate variability into aspects [18]. Indeed, each language (`EN`, `FR`, `DE` in Figure 2) is considered as a variant. The behavior of the advice can be described as follows. The `I18N` interface provides a set of methods responsible for translating a pre-defined set of labels. It also provides a look-up method `get(String myString): String` that returns, if possible, the translation of `myString` for a given language. In a component (*e.g.*, `FR`), if the look-up method cannot translate a word, the component ask the dispatcher to find a translation for this word. The dispatcher will ask the components according to a predefined policy (*e.g.*, `EN` first if available). If the word exists in the local database, it is translated (*e.g.*, from English to French) thanks to the translator that sends a request to a website dedicated to translation. Note that in the advice, all the components and bindings are unique (depicted with a '1' in the Figure). This means that even if there exist multiple join points and/or even if the aspect is applied several times, these elements will only be woven once in the base model. In the composition protocol, the bindings are not unique and will be introduced for all the identified join points. The pointcut simply identifies any component that requires the `I18N` interface, with no more assumption. The weaving process has 2 steps: matching (or join point detection) [31] and composition. In our example, the matching step detects all the components that require the `I18N` interface. Then, the composition step binds the `I18N` server interface provided by the advice to the client interfaces of the join points. When an aspect is being composed, the inverse composition protocol is automatically generated. It allows us to easily unweave an aspect when it is not longer adapted to the context.

After some aspects have been woven into the source model, the target model we obtain is validated, as illustrated in Figure 1, step 3. We will present in more details this validation step in Section 3.

2.3 On-the-fly Generation of Reconfiguration Scripts

As mentioned by Zhang and Cheng in [35], if there exists N possible configurations, this may lead to N(N-1) possible transitions. If N is large, it rapidly becomes difficult to handle these transitions by hand.

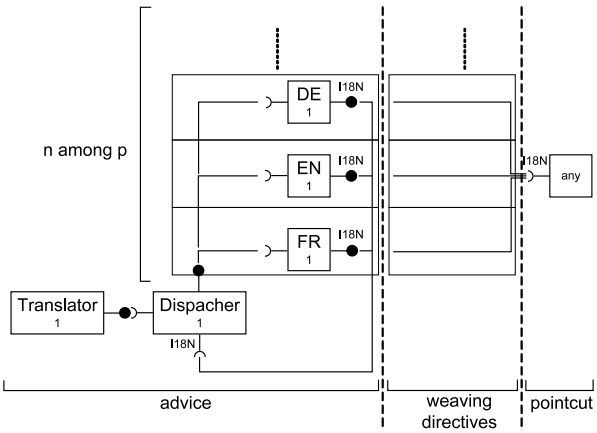


Figure 2. I18N aspect

In a traditional model-driven development, models are refined and transformed step by step down to source code. If some changes appears in the requirements, it is possible to propagate these changes to the models and regenerate the source code in order to rapidly propose a new version of the system. In the context of adaptive systems, it is not possible to adopt this schema. Indeed, such systems should offer continuous services and cannot be stopped, regenerated and restarted. The system should keep executing while being reconfigured from one configuration to another.

Once a target model (representing the system we want to reach) is created and validated, it is compared with the source model (representing the actual architecture of the running system). In the current implementation of our tool, we use EMF Compare[1] in order to compare models. It produces a diff and a match model that specifies the differences and the similarities between the source and the target models, as illustrated in Figure 1, step 4. The comparison engine is generic, so it is possible to compare any kind of models. The algorithm is quite similar to the algorithm proposed by Nejati *et al.* in the context of statechart specifications [28]. It considers the properties of each model element as well as its neighbors in order to compute a similarity degree. Note that it is possible to customize the comparison engine to consider the specificity of a given domain metamodel. However, the generic engine provides sensible results for our metamodel describing runtime architecture, with no customization.

Then, we automatically analyse both diff and match models to obtain the relevant changes between the source model and the target model *e.g.*, addition/removal of components/bindings, changes of attribute values, etc. However, it is not possible to adapt the running system during this analysis. For example, if the model comparison detects a component removal before a binding removal, directly adapting the system would lead to a dangling binding that might not be allowed by the underlying execution platform.

In order to tackle this issue, we reify each significant modification as a reconfiguration command during the analysis (Figure 1, step 4). Each command implements an atomic platform-specific reconfiguration (adding and/or removing bindings and/or components, etc.) and declares a priority. We first stop the components that have to be stopped and then remove bindings before removing components. We add components before adding bindings and finally restart the components that should be restarted. When the analysis of the model comparison is achieved, we execute the ordered sequence of commands to actually adapt the running system in a safe way (Figure 1, step 5). This set of commands is the transition that transforms the source system into the target system. Depending on the execution platform we use (*e.g.*, OSGi, Fractal or OpenCOM) a factory will instantiate the corresponding commands.

3 Validating Target Configurations

In Section 2, we showed how we can obtain configurations by weaving some aspects into a base configuration. Using aspect models, instead of directly adapting the running system using low-level reconfiguration scripts [10] allows us to reason more easily and help designers in identifying interactions between aspects [13]. More details about aspect interaction detection are given in Section 3.1.

Then, we showed how to generate reconfiguration scripts to make the running system evolve from a source configuration (the current configuration), to a target configuration. However, in the context of adaptive systems, we should ensure that the target configuration we want to reach makes sense. This is why we do not directly reflect the changes appearing in the source model to the running system. When the number of configurations is limited, for example in the case of critical embedded systems, it is possible to validate at design time all the possible configurations [35]. However, in larger-scale adaptive systems, this systematic validation may become too time and resource consuming to be realistic. Moreover, once the system has been deployed, new variation points that have not been foreseen may appear while the system is running and cannot be stopped. In this case, it is very useful to validate configurations on the fly before actually adapting the

[1] See http://www.eclipse.org/modeling/emft/

running system. This is detailed in Section 3.2.

3.1 Detecting Aspect Interactions

In Section 2 we introduced an internationalization aspect. Most of the aspects of our case study (see Section 4) and most of the components of the base system (for example, GUI) also needed this aspect. Weaving the I18N aspect before the other aspects may cause some important messages not to be translated. Using techniques like Critical Pair Analysis (CPA) allows us to detect interaction between aspects [13]. Basically, if the pointcut of an aspect *A1* can be matched in the advice of another aspect *A2* it means that *A2* introduces some join points for *A1*. In other words, *A1* should be woven after *A2* in order to be able to consider newly introduced join points.

For example, let us introduce a second aspect responsible for preventing devices deployed in the house (lights, electric shutters, etc) to be damaged due to too many successive transitional regimes. This aspect, called **event filter**, will ignore all the antagonist actions that appears in a too short period. All the events sent to the unstable device controller by the device proxy are derived into the event filter component. These events are cached during a given period that depends on the type of the device. Events are delegated to the device if no antagonist events appeared during the cache period. These filters cannot be systematically deployed as they make devices less reactive in standard conditions. This is why filters should be dynamically and locally deployed and undeployed. Every canceled actions has to be logged and displayed in a language the user understand, using the I18N interface. As the event filter aspect introduces a component that requires the I18N interface, it introduces a new join point for the internationalization aspect. Consequently the I18N aspect should be woven after the event filter aspect.

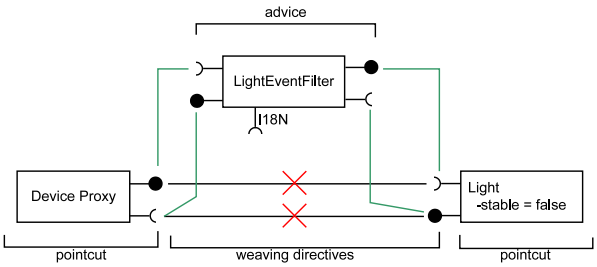


Figure 3. Event Filter aspect

This kind of basic analysis can help designers in identifying interactions that cannot easily be detected when directly working with low-level reconfiguration scripts. However, CPA has limitations. For example, this kind analysis is not associative. If no interaction exists between *A1* and *A2* and no interaction exists between *A1* and *A3*, nothing ensures that there exists no interaction in the triplet {*A1*, *A2*, *A3*}, depending on the weaving order. Determining the interactions that may exist in all the possible subsets of aspects is very complex as the number of combinations grows rapidly.

3.2 Validating Target Configurations

As explained in the introduction of this section, it is not always possible to check all the possible configurations of an adaptive system *a priori*, for time and resource issues. Moreover, the apparition of unforeseen adaptation while the system is already deployed makes it impossible to perform all the validation process at runtime. However, in the context of high-insurance adaptive systems [20] any configuration we wanted to reach should be validated.

In order to validate target configurations (Figure 1, step 3), we propose to define some invariants on the metamodel we use to represent runtime architecture and check these invariants for every constructed (by aspect weaving) target configuration. These invariants are expressed as Kermeta [26] meta-aspects that are woven into the metamodel we are using for representing runtime architecture. Kermeta meta-aspects can be used to refine existing meta-classes by integrating contracts (pre/post-conditions, invariants), attributes and references, operations and super-classes. The invariant illustrated in Figure 4 specifies that all the client and non optional ports defined in the component type of the component should be bound. In other words, it detects if mandatory bindings are missing. It uses an OCL-like syntax[2] which provides high-level operators for navigating and querying models: *select*, *exist* and *forall* in our example. Another invariant checks that the server interface of a binding is a sub-type of the client interface. Currently, six invariants are woven into our metamodel. Note that end-user can define more specific invariants to ensure the validity of the configurations.

Invariant checking, as well as the steps related to aspect weaving and aspect interaction detection, can be performed on a tiers system, independent from the running system itself. Indeed, all the models (metamodel, configurations and aspects) can be serialized in XML and transmitted to other systems.

[2]Kermeta now allows to define constraints using the real OCL syntax

```
1  aspect class Component {
2    inv optionalClientPortBound is do
3      self.type.ports.select{p |
4          not p.isOptional and
5          p.role == PortRole.CLIENT
6      }.forAll{p |
7          self.binding.exists{b |
8            b.client == p
9          }
10     }
11   end
12 }
```

Figure 4. Checking mandatory bindings

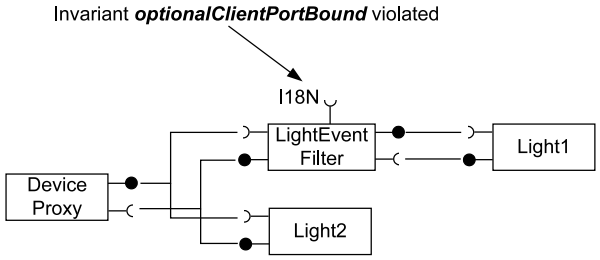


Figure 5. Invariant violated

Another possible solution to validate target configuration before actually adapting the running system would be to simulate models. This can be done by describing the behavior of each configurations, for example using state machines or Petri nets [35]. Then, it would be possible to use Kermeta [26] to execute these models and perform the simulation and detect deadlocks, for example. However, in order to manage the explosion of variants, this behavior should be associated to each aspect and should be composable in order to obtain the behavior of whole configurations. SmartAdapters is well adapted to compose structural aspects, in class or component diagrams. However, to consider the semantic of behavioral models, it has to be customized by hand. Another solution would be to use a specific aspect weaver dedicated to the composition of behavioral models. Such an approach is presented in Section 5 and its integration with our approach is discussed in perspectives.

If the target configuration we want to reach is valid, then the process continues, as illustrated in Figure 1, step 4, in order to actually adapt the running system. If the target configuration is not valid, then our rollback mechanism simply consists in discarding this target configuration and do not submit it to the following steps of the process. Indeed, as the modification on the model are not directly reflected to the running system, we do not have to cancel platform-level reconfigurations. An error report is automatically raised by Kermeta [26], specifying which invariants are violated by the target configuration. This helps the system or the user in understanding why the configuration is not valid. For example, if we consider that the I18N client port is mandatory, then we are able to detect the cases where the I18N has been woven before (or not at all) other aspects, without using the preliminary critical pair analysis. Indeed, the invariant illustrated in Figure 4 detects missing mandatory bindings. This case is illustrated in Figure 5 that shows a fragment of the base model where the event filter has been woven and the I18N is not woven at all.

4 Application to Home-Automation

4.1 EnTiMid to help people to stay at home

Industrials, associations and public institutions of the metropolis of Rennes, are working together on a project which aims to allow dependent people to stay at home as long as possible. Due to the large scale of the project, and the diversity of disabilities that have to be considered, the deployment context will be different for each equipped house. The technologies used will vary, in order to compensate handicaps or because a technology is already installed, and people do not want it to be removed. Moreover, the system installed in these houses will have to provide a remote access to the devices of the house, and transmit all the necessary information from the sensors of the house to a control center where information will be treated. Those access and transmissions can be realized through various ways (Internet, POTS, SMS) and the medium used will vary according to the availabilities.

An abstraction layer over all these devices has been developed in the form of a multi-facet middleware called EntiMid [27]. Based on an OSGi platform [33], EnTiMid is composed of different components (called bundles), that can be dynamically added, removed, started or stopped, with no need to restart the entire system. Each of them can offer or require services to/from other ones, but such services can disappear at any time. One of those services, identified as *TechProvider*, aims at translating the information caught from a device network protocol, into EnTiMid standard event messages, and vice versa. By this way, high level services can manage devices through unified messages, whatever the underlying technology is. EntiMid

allows engineers to prescribe the most adapted technology, with no regard to the communication technology it uses. Then high level services can be developed to offer different kind of services to inhabitants and health professionals. For example, automatic energy, heating, access or light management to ease the everyday life; remote control and alert transmissions, to allow professionals to intervene on the house, in a short time, according to the information collected.

A show apartment will soon be available, and EnTiMid will be deployed in order to test its functionalities with devices installed by industrials. Those real conditions will have for consequence an identification of new needs of development and variability.

4.2 Designing Variability Dimensions

We now introduce and justify the need for dynamic variability in our case study. Each variability dimension is represented by a set of aspects designed with SmartAdapters.

Device Management. Physical devices are managed by EnTiMid. Most of the devices are installed when the system is deployed. However, new physical devices may be installed after the initial deployment and consequently, they should be managed dynamically while maintaining the functionalities offered by already deployed devices. In our case study, we have to manage 6 lights, 4 heaters, 3 mixing valves (controlling water temperature) and 3 electric shutters. Moreover, depending on the evolution of the patient, devices should be managed in different ways. For example, in the case the patient becomes visually impaired, the power of the lights should be increased by 10%. Another focus can be the evolution of a mobility handicap. In a first stage, the patient can move alone in the house, allowing him to manually close the shutters. Then the evolution of the disease makes it difficult for the patient to get out of his bed. A remote control will then be offered to this person to ease his everyday life, and the control system of the house have to adapt to this situation.

Each low-level protocol (KNX[3], X10, X2D[4], etc) manages devices in an ad-hoc way. Consequently, we would have to define a variability dimension for each protocol. For the sake of clarity, we present one generic variability dimension that harmonizes the concepts present in each low-level protocol. It is illustrated in Figure 6. In order to represent this variability dimension, we leverage the ability of SmartAdapters to integrate variability into aspects [18]. Each type of device controller (Light, Heater, Shutter and Mixing Valve) is a variant. Each type of component can be instantiated several times. Each device also offers an interface for loading pre-defined scenarios. All the device controllers are connected to a device proxy that receives messages from the EnTiMid platform and dispatches these messages to the appropriate device.

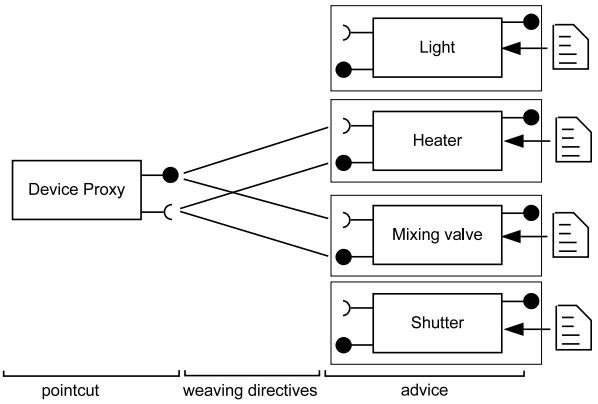


Figure 6. Device Management aspect

Permission Management. All the physical devices in the house are supposed to be potentially controlled by EnTiMid. However, doctors can choose for each device whether it is controlled by the system or by the patient, according to the degree of autonomy of the patient. If the device has no permission manager, the patient can interact with the device. With a permission manager, the patient cannot interact with the device that only follows a pre-defined scenario.

The permission management aspect illustrated in Figure 7. In order to control the access from the user, an aspectual component [29] intercepts, using an AspectJ-like pointcut, every call to the services of the controller, log each attempt and does not proceed. The device will simply execute its pre-defined scenario without being interrupted.

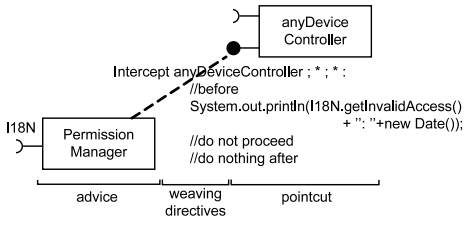


Figure 7. Permission Management aspect

Two other aspects (internationalization and event filter) have already been introduced in previous sections. Table 1 summarizes the number of aspects we

[3]http://www.knx.org/
[4]http://www.english.deltadore.com

really need for each variability dimension to implement our case study.

Dimensions	Aspects	Aspect variants
Device Mgmt	3 (KNX, X10, X2D)	4 (1 per device type)
Permission Mgmt	1	0
Event Filter	3	4
I18N	1	10
Total	8	18

Table 1. Number of aspects per variability dimension

4.3 Constructing a Configuration by Aspect Weaving

We are now going to propose to illustrate the weaving process with some of the aspects we have identified in the previous sub-section into our motivating example (Figure 8). The top of the figure illustrates a snippet of the base configuration. It allows to access low-level devices using two high-level protocols: UPnP[5] and DPWS[6], via the EnTiMid component. In the bottom part of the figure three aspects have be woven: a filter aspect that filters the events send to Light1 ; two permission managers that restrict the heater and the shutter to their pre-defined scenarios. Finally, the internationalization aspect is woven. The interaction between the aspects can be observed in the example: the event filter and the permission manager aspects uses the internationalization aspects.

With the five variability dimensions we have defined earlier in this section, we can obtain a wide range of possible configurations. If we consider the three aspects that directly impact devices (device management, permission management and event filter), we obtain five possible modes for each device, as shown in Table 2, where 0 indicate that the aspect is not active for the device. As the devices are managed independently, this leads to $5^{16} \approx 15 \cdot 10^{10}$ possible configurations. This number is even greater if we consider the internationalization aspect.

If we consider the $15 \cdot 10^{10}$ possible configurations, we obtain approximately $225 \cdot 10^{20}$ possible transitions from one configuration to another. The configurations are obtained on-demand: the weaving of some aspects can be triggered by hand depending on the choices of a human operator (doctor or technician), while some other aspects (*e.g.*, event filter) are triggered by the context. Before adapting the system, all the invariants

[5]http://www.upnp.org/

[6]http://en.wikipedia.org/wiki/Devices_Profile_for_Web_Services

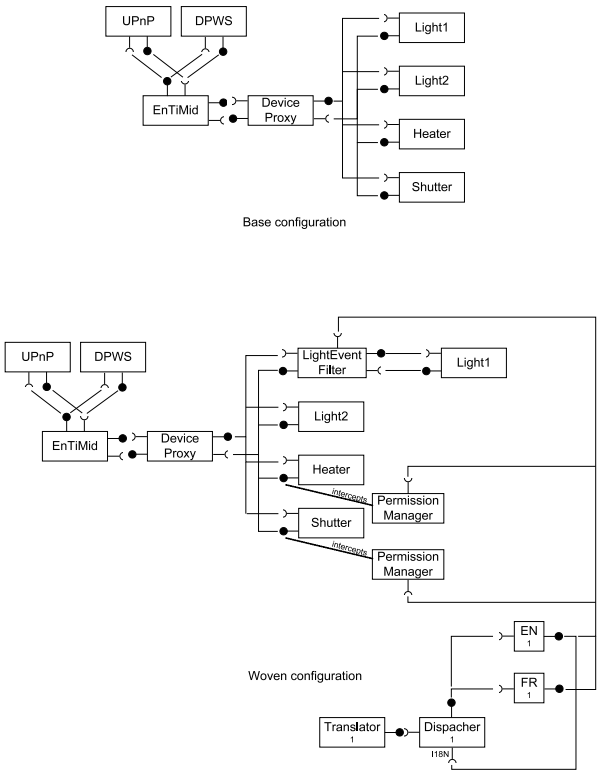


Figure 8. Base and woven configurations

Device Mgmt.	Permission Mgmt.	Event Filter
0	0	0
1	0	0
1	0	1
1	1	0
1	1	1

Table 2. Five Operating Modes per Device

defined in the metamodel are checked on the woven target model. This allows us to determine whether the target model is well-formed or not. If the configuration is valid, the transition toward the target model is automatically computed using model comparison. If we consider the internationalization aspect, we should multiply the number of configurations by 2^{10} as we should handle at least one language among 10. Table 3 summarizes the number of dimensions we have defined, the total number of aspects we need, the number of configurations we obtain on-demand by aspect weaving and the number of transitions we generate on-demand when moving from one configuration to another.

Dimensions	Aspects	Configurations	Transitions
4	8	$> 15 \cdot 10^{13}$	$> 225 \cdot 10^{26}$

Table 3. Number of configurations and transitions managed by aspects

5 Related Work

In [35], Zhang and Cheng propose to adopt a model-driven approach for designing and validating dynamically adaptive software systems. This approach focus on the behavior of adaptive systems whereas we mainly focus on the architecture of running systems. In [35], the behavior is modeled with state-based diagrams like Petri nets. Zhang and Cheng define an adaptive system as a set of simple adaptive systems. A simple adaptive system is defined by three entities: a source system, a target system and transitions responsible for moving the source system to the target system. All the source systems, target systems and transitions should be explicitly modeled, leading to an explosion of artifacts needed to manage an adaptive system. However, this exhaustive representation allows validating intensively the system at design time. Finally, using code generation, adaptive programs are derived from the models. In our approach, configurations are constructed on demand by selecting and weaving a set of aspects. Once composed, it is possible to check the target configurations we want to reach. Then, the transitions needed to adapt the system is automatically generated using model comparison.

In [10], David *et al.* present SAFRAN (Self-Adaptive FRActal compoNents) an Aspect-Oriented approach for implementing self-adaptive system on the Fractal [7] platform. In order to adapt the system, they define adaptation policies, separately from the business logic, which follow this pattern: **when** <*event*> **if** <*condition*> **do** <*action*> where actions are low-level reconfiguration scripts. Batista *et al.* [3] propose the same kind of approach for the OpenCOM [6] execution platform. These approaches do not propose an explicit representation of the target configurations. Consequently, it is not possible to easily visualize the system nor to perform validation, simulation before actual adaptation. In our approach, configurations are obtained by weaving aspects into a model representing the current system. Woven configurations are checked against invariants. We then generate all the adaptation logic needed to adapt the system while these approach have to specify low-level reconfiguration scripts. Moreover, our approach is not specific to a given execution platform. Finally, script-based approach do no offer support for easily determining interactions between reconfiguration scripts while Aspect-Orientation provides some mechanisms [13].

Ensuring software correctness is an important issue and this is amplified when dealing with software variation . Correctness is even more important in Dynamically adaptive System where variation is handled at runtime. This issue has been addressed by the software product lines community [30, 32]. One of the main concerns for correctness in product lines is about the methods to be used in order to limit the number of tests to be performed for a family of products. Two issues are especially considered: the increase of work for the programmer and the time spent to perform them [8, 19]. These contributions mainly introduce formal methods in order to exploit the commonalities of a software family in order to achieve these issues. They rely on SAT solver [9] or more generally on model-checking [17] techniques in order to verify those tests. In our case, the main difference is the fact that verifications can only be done at runtime. New aspect can be designed and integrated, consequently unanticipated evolution can occur. Using MDE techniques allows software developer to apply his aspect on an abstraction of its runtime system to check its correctness.

In [24], we present a first approach that combines AOM and MDE in order to manage variability at runtime. In this paper, we show how aspects can help designer in determining interactions between dynamic variants and how models allows to validate new configurations independently from the running system and easily roll-back when a configuration is not valid.

In [34], Wolfinger *et al.* demonstrate the benefits of integrating Software Product Line techniques to manage the runtime reconfiguration and adaptation mechanisms on the .NET platform. Automatic runtime adaptations are attained by using the knowledge documented in variability models. As many authors [1, 2, 22] advocate that aspect-oriented software development (AOSD) is an effective technique to support feature variability, this approach is close to ours. Automatic runtime adaptations are attained by using the knowledge documented in variability models. However, they do not propose to do a preview of the running system at the model level to check its correctness.

In the domain of Aspect-Oriented Modeling, Nejati *et al.* [28] propose an approach for matching and merging statechart specifications. This approach would be useful for extending our approach with behavior, as mentioned in Section 3.2. If we describe the behavior of our aspects it would possible to merge this behavior with the base model in order to obtain the complete be-

havior. Describing the behavior of our aspects mainly consists in modeling the behavior of the interfaces of each component and compose these behavioral models when components are assembled. Once we obtain the global behavior, it is possible to reuse the concepts proposed by Zhang and Cheng [35] for validating the behavior of adaptive systems.

6 Conclusion

In this paper, we have presented our approach for managing the complexity of dynamically adaptive systems. This approach combines aspect-oriented and model-driven techniques in order to limit the number of artifacts needed to realize dynamic variability. Our aspect model weaver allows us to construct configurations on-demand by selecting, by hand or according to predefined conditions, a set of aspects. Using the woven configuration, it is possible to validate this configuration before actually adapting the running system. Using aspects instead of low-level reconfiguration scripts allows us to detect some interactions that can provide assistance when selecting the set of aspects to be woven. Then, target configurations obtained after aspect weaving are checked with respect to the invariant we have defined into our metamodel. If a target configuration is not valid, the roll-back mechanism simply consists in not submitting this target configuration to the sub-sequent steps of the adaptation process. If the configuration is valid, we generate the adaptation logic using model comparison. This allows us to automatically determine a safe transition to make the system evolve from a its current configuration to the target configuration.

In future works, we plan to extend our approach following different axis. Currently, we describe our systems according to their runtime architecture (components, bindings, etc). We will also consider the behavior of dynamically adaptive systems. This can be realized if we can modularize and compose the behavior of components. Thus, it would be possible to decompose the system behavior into aspects, as we do for the architecture. We plan to reuse the approach we have presented in the related work section to consider the behavior. Another axis is about the validation of target configurations. Currently, we ensure that the target configurations ensure the invariants defined in our metamodel. With the definition of the behavior, we would be able to perform simulation in order to detect some deadlocks, for example.

References

[1] V. Alves, P. Matos Jr, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming. *Transactions on Aspect-Oriented Software Development IV*, 4640/2007, 2007.

[2] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: 28th International Conference on Software Engineering*, pages 122–131, Shanghai, China, 2006. ACM.

[3] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *EWSA'05: 2nd European Workshop on Software Architecture*, pages 1–17, Pisa, Italy, 2005.

[4] N. Bencomo, G. Blair, and R. France. Models@run.time (at MoDELS) workshops. www.comp.lancs.ac.uk/ bencomo/MRT/.

[5] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *ICSE'08: Formal Research Demonstrations Track*, Leipzig, Germany, 2008.

[6] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. Opencom v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, USA, 2004.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL Component Model and its Support in Java. *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

[8] M. Cohen, M. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 53–63, Portland, Maine, 2006. ACM.

[9] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *GPCE'06: 6th Int. Conf. on Generative Programming and Component Engineering*, pages 211–220, Portland, Oregon, USA, 2006. ACM.

[10] P. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *SC'06: 5th Int. Symposium on Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97, Vienna, Austria, 2006.

[11] E. Figueiredo, N. Cacho, C. SantAnna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE'08: 30th International Conference on Software Engineering*, pages 261–270, Leipzig, Germany, may 2008. ACM.

[12] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel. Modeling and Validating Dynamic Adaptation. In *3rd International Workshop on Models@Runtime (MODELS'08)*, Toulouse, France, oct 2008.

[13] P. Jayaraman, J. Whittle, A. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07: 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[14] J.-M. Jézéquel. Model Driven Design and Aspect Weaving. *SoSyM'08: Journal of Software and Systems Modeling*, 7(2):to appear, march 2008.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[17] T. Kishi, N. Noda, and T. Katayama. Design Verification for Product Line Development. In *SPLC'05: 9th International Software Product Lines Conference*, volume LNCS 3714, pages 150–161. Springer, 2005.

[18] P. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07: 10th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Nashville USA, Oct. 2007.

[19] M. Mannion and J. Cámara. Theorem Proving for Product Line Model Verification. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003*, volume LNCS 3014, pages 211–224. Springer, 2003.

[20] P. McKinley, B. H. C. Cheng, C. Ofria, D. Knoester, B. Beckmann, and H. Goldsby. Harnessing digital evolution. *Computer*, 41(1):54–63, 2008.

[21] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

[22] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT'04/FSE-12: 12th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 127–136, Newport Beach, CA, USA, 2004. ACM.

[23] B. Morin, O. Barais, J. M. Jézéquel, and R. Ramos. Towards a Generic Aspect-Oriented Modeling Framework. In *3rd Int. ECOOP'07 Workshop on Models and Aspects, Handling Crosscutting Concerns in MDSD*, Berlin, Germany, August 2007.

[24] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In *MoDELS'08: 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[25] B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jzquel. Managing Variability Complexity in Aspect-Oriented Modeling. In *MoDELS'08: 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Toulouse, France, October 2008.

[26] P. Muller, F. Fleurey, and J. M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer. Kermeta is available at: http://www.kermeta.org/.

[27] G. Nain, E. Daubert, O. Barais, and J. M. Jézéquel. Using MDE to Build a Schizonfrenic Middleware for Home/Building Automation. In *ServiceWave'08: Networked European Software & Services Initiative (NESSI) Conference*, Madrid, Spain, december 2008.

[28] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *ICSE'07: 29th International Conference on Software Engineering*, pages 54–64, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[29] N. Pessemier, L. Seinturier, T. Coupaye, and L. Duchien. A Component-Based and Aspect-Oriented Model for Software Evolution. In *IJCAT'07: International Journal of Computer Applications in Technology, Special Issue on Concern-Oriented Software Evolution*, volume 4089 of *Lecture Notes in Computer Science*, page 259273, Vienna, Austria, mar 2006. Springer-Verlag.

[30] K. Pohl and A. Metzger. Software Product Line Testing. *Commun. ACM*, 49(12):78–81, 2006.

[31] R. Ramos, O. Barais, and J. M. Jézéquel. Matching Model Snippets. In *MoDELS'07: 10th Int. Conf. on Model Driven Engineering Languages and Systems*, page 15, Nashville USA, Oct. 2007.

[32] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. In *International Workshop on Software Architectures for Product Families*, volume LNCS 1951, pages 146–157, Spain, 2001. Springer.

[33] The OSGi Alliance. OSGi Service Platform Core Specification, Release 4.1, May 2007. http://www.osgi.org/Specifications/.

[34] R. Wolfinger, S. Reiter, D. Dhungana, P.Grunbacher, and H. Prahofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *ICCBSS'08: 7th Int. Conf. on Composition-Based Software Systems*, pages 21 – 30, 2008.

[35] J. Zhang and B. H. C. Cheng. Model-based Development of Dynamically Adaptive Software. In *ICSE'06: 28th International Conference on Software Engineering*, pages 371–380, Shanghai, China, 2006. ACM Press.

MODELS@ RUN.TIME TO SUPPORT DYNAMIC ADAPTATION

Brice Morin, *INRIA*
Olivier Barais and Jean-Marc Jézéquel, *INRIA and IRISA, University of Rennes*
Franck Fleurey and Arnor Solberg, *SINTEF ICT*

An approach for specifying and executing dynamically adaptive software systems combines model-driven and aspect-oriented techniques to help engineers tame the complexity of such systems while offering a high degree of automation and validation.

Today's society increasingly depends on software systems deployed in large companies, banks, airports, and so on. These systems must be available 24/7 and continuously adapt to varying environmental conditions and requirements. Such *dynamically adaptive systems* exhibit degrees of variability that depend on user needs and runtime fluctuations in their contexts. Engineers can develop DASs by defining several variation points. Depending on the context, the system dynamically chooses suitable variants to realize those variation points. These variants may provide better quality of service (QoS), offer new services that did not make sense in the previous context, or discard some services that are no longer useful.

DASs range from small embedded systems to large systems of systems and from human-driven to purely self-adaptive systems. A DAS can be conceptualized as a *dynamic software product line* in which variabilities are bound at runtime.[1] Similar to traditional SPLs,[2] the number of possible configurations of a DSPL grows combinatorially with the number of variation points and variants. In an SPL, products are derived by human decisions and are totally independent; a DSPL also must handle the migration paths between those configurations.

The configurations produced by a DSPL are thus highly dependent: The system should evolve at runtime between its current configuration (source) and a new configuration (target) through a safe migration path (transition). Several stimuli trigger these transitions: context changes, user preferences, and so on. The DSPL must provide support for describing the adaptation logic.

A DSPL's execution can be abstracted as a highly connected state machine,[3,4] where the states are the possible system configurations and the transitions the migration paths. Fully specifying this state machine lets the designer perform extensive simulation, validation, and testing of the system's dynamic variability before actually implementing the system.[5] Model-driven engineering (MDE) techniques then make it possible to fully generate the adaptive system's code[4] from the state machine specification.

 Published by the IEEE Computer Society 0018-9162/09/$26.00 © 2009 IEEE

However, this approach suffers from two main drawbacks related to adaptation management and evolution management:[6]

- *Explosion in the number of artifacts.* Even when the designer specifies the state machine at a high level of abstraction, the number of configurations and transitions to be described grows rapidly. For example, in combining the features of one dynamic customer relationship management (DCRM) system, we counted 92,160 configurations;[7] this leads to 92,160 × 92,159 = 8,493,373,440 possible transitions and triggers among these configurations. Validating all these artifacts can soon become a problem: The number of configurations explodes in a combinatorial way with regard to the number of variants, and the number of transitions is quadratic with regard to the number of configurations. While it is still possible to specify this state machine for simple adaptive systems, this rapidly becomes a daunting task in the case of large systems comprising a wide range of variation points.[8]
- *Evolution of the adaptive system.* Once the adaptive system is deployed and running, it can evolve based on new user needs, detection, and correction of limitations or security weaknesses. Evolving an adaptive system involves dynamically changing the adaptation state machine: adding and removing states and transitions. Applying a classic MDE approach to generate all the application code from higher-level specifications would be impractical: The system would have to be stopped and decommissioned before a new system—based on modifications to the state machine—could be deployed and started. This would make the system unavailable for a long period, which in many cases would be unacceptable.

In our work on the EU-ICT DiVA project (Dynamic Variability in complex, Adaptive systems; www.ict-diva.eu), we address these two drawbacks by using software models at runtime as well as at design time. We rely on four metamodels supported by design tools, such as graphical or textual editors and validators or simulators, to assist in modeling the DSPL at design time. Models conforming to these four metamodels are the main data manipulated by the runtime infrastructure responsible for dynamically adapting component-based applications at runtime. These models provide a high-level basis for reasoning efficiently about relevant aspects of the system and its environment and offer enough details to fully automate the dynamic adaptation process. It is possible to make the design specifications evolve at any time, before initial deployment or while the system is already running.

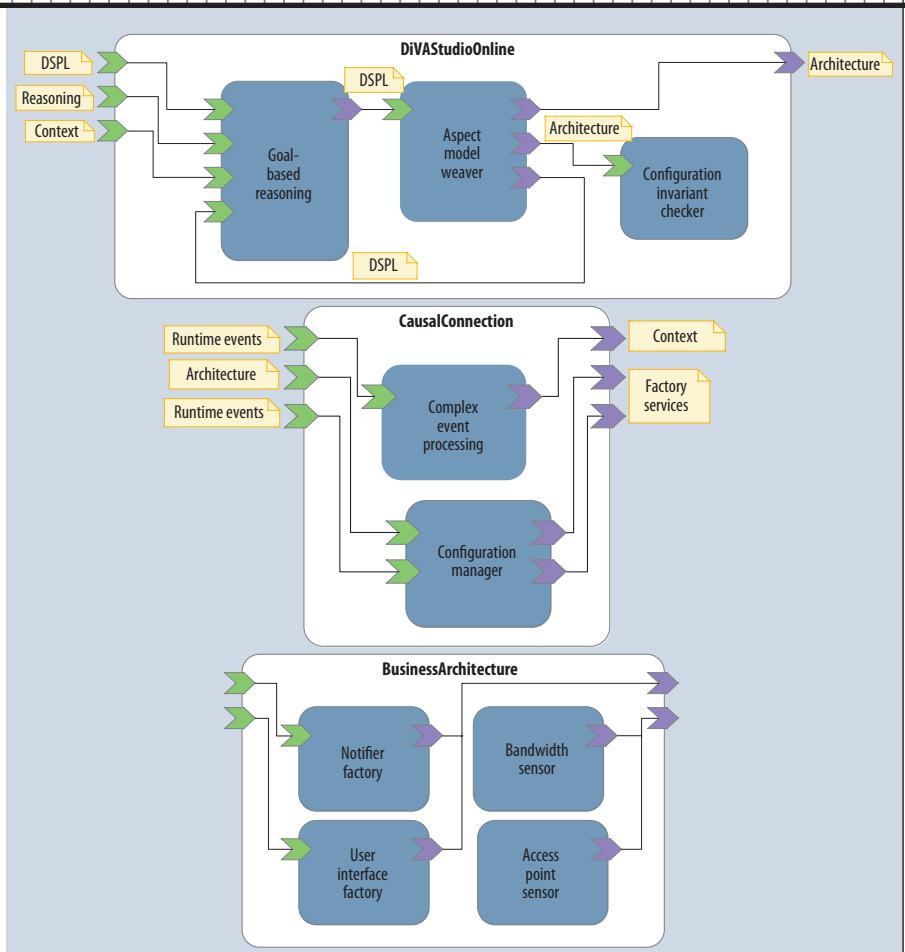


Figure 1. Runtime architecture to support dynamic software product lines.

MODEL-ORIENTED ARCHITECTURE

Figure 1 shows the architecture for managing DSPLs at runtime, comprising three layers:

- *DiVAStudioOnline*, a platform-independent layer that only manipulates models;
- *CausalConnection*, a platform-specific layer that links the model space to the runtime space; and
- *BusinessArchitecture*, an application-specific layer that

DESIGNING FEATURES AS ASPECT MODELS

In DiVA, we leverage aspect-oriented modeling (AOM) techniques to refine features and automatically build complete configurations before the actual adaptation. A base model refines the system's commonalities—elements present in all the configurations—as an architecture made of components and the connections between them, or bindings. Aspect models refine the system's variants by specifying their precise architectures: Each model is an architectural fragment that contains all the information needed to be easily plugged into the base architecture.

As Figure A shows, an aspect model consists of three parts:

- *Advice model.* This architectural fragment specifies what is needed to realize the associated variant. An advice model need not be fully consistent. The base model should bring the missing elements needed to make the advice model consistent when weaving the aspect.
- *Pointcut model.* This architectural fragment specifies the components and bindings that the aspect model expects from the base model to be woven—that is, where the aspect should be woven. The most precise is the pointcut model, the smallest is the set of potential places where the aspect can be woven, and vice versa. For example, if a component's type is not specified in the pointcut model, this component would be matched by any of the base architecture's components, irrespective of its real type.
- *Composition protocol.* This describes how to integrate the advice model into the pointcut model. When weaving the aspect, the places matching the pointcut model automatically contextualize the composition protocol to actually weave the aspect into the base model.

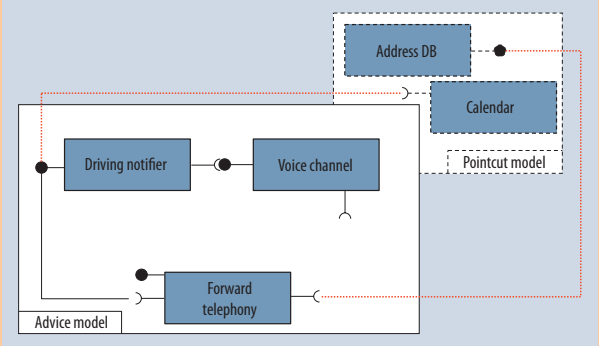


Figure A. Aspect model. An advice model specifies what is needed to realize the associated variant, a pointcut model specifies the components and bindings that the aspect model expects from the base model to be woven, and a composition protocol describes how to integrate the advice model into the pointcut model.

Refining features as aspect models allows the designer to validate the adaptive system one step further. Since AOM relies on a strong theoretical background, such as graph theory, it is possible to perform analysis—for example, critical pair or confluence analysis—to detect previously undetected aspect dependencies and interactions.[1] The designer can update the DSPL and reasoning models' constraints to avoid invalid interactions. It is also possible to refine the simulation step by actually producing, via aspect weaving, and validating some detailed configuration corresponding to foreseen contexts.

Reference

1. P. Jayaram et al., "Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis," *Proc. 10th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 07), LNCS 4735, Springer, 2007, pp. 151-165.

defines factory components, which simply provide services to instantiate multiple component instances. This layer is not part of the infrastructure for managing DSPLs.

Five components—a complex event processor, a goal-based reasoning engine, an aspect model weaver, an online configuration checker, and a configuration manager—interact by exchanging models conforming to the metamodels.

Metamodels

The components exchange four kinds of metamodels: DSPL, context, reasoning, and architecture.

DSPL. This is a feature model that describes the system's variability. Commonly used in the SPL community,[2] feature models describe hierarchies with mandatory features, options, alternatives—*n* among *p* choices, and so on—as well as constraints (requires, excludes) among features. The DSPL model is a regular feature diagram model, with a naming convention to refer to architectural fragments, or *aspect models*, refining the features. In this way, engineers can exploit any existing feature model tools—graphical editors, checkers, and others—with no modification.

Context. This model specifies the system's environment. A set of context variables specifies those aspects of the environment relevant to adaptation. At runtime, the variable values are provided by context sensors, and these may trigger a system reconfiguration.

Reasoning. This model describes selection of the DSPL's features according to context. Several formalisms exist such as event-condition-action (ECA) rules[9] or goal-based optimization rules.[7] We do not impose any particular reasoning model. An ECA model will typically describe, for particular contexts, which features to select. A goal-based model will typically describe how features impact QoS properties—using, for example, help and hurt relationships[10]—and specify when QoS properties should be optimized, for example, when a property is too low.

Architecture. This model describes component-based architectures. Designers can use any metamodeling framework, such as the Unified Modeling Language (UML) or Service Component Architecture (SCA), or any architecture description language, to describe the architecture. Because our dynamic adaptations are currently reconfigurations, our focus is on components and bindings, concepts that are present in metamodels. In practice, we have defined our own minimal metamodel[11] to reduce memory overhead at runtime. Other metamodels map to our metamodel via model transformations in Kermeta (www.kermeta.org).[12]

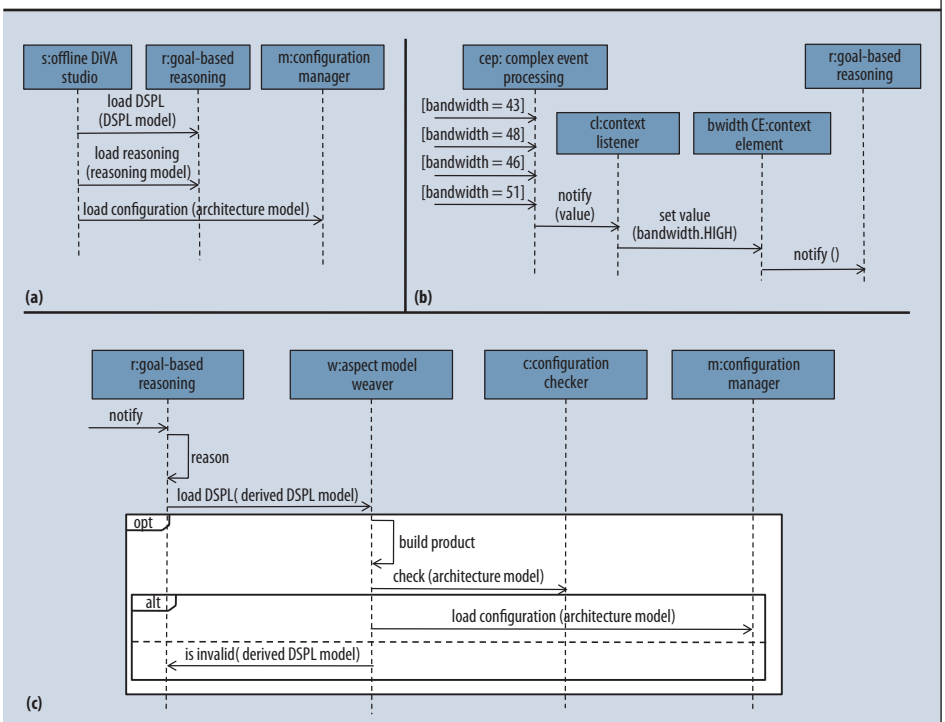


Figure 2. Interactions between the components: (a) initialization; (b) monitoring; (c) reasoning, derivation, and dynamic adaptation.

The architecture model refines each leaf feature of the DSPL model into an architectural fragment. As the "Designing Features as Aspect Models" sidebar describes, we use aspect-oriented modeling (AOM) techniques to design and compose features into a core model containing the mandatory elements.

Designing the models

Engineers design these models offline before the initial system deployment or while the system is already running, but independently of the running system, and leverage them at runtime to drive the dynamic adaptation process. The quality and correctness of models conforming to these metamodels are crucial and must be checked as early as possible. Since the components of the DiVAStudioOnline layer exchange only models conforming to these four metamodels, it is very easy to validate the adaptation logic: A test component simply produces a set of input models, and another test component analyzes the models produced by the DiVAStudioOnline components. For example, a test component provides a DSPL model, a reasoning model, and a scenario (sequence of context models) to simulate the system's adaptation logic,[5] and another test component checks the produced configurations.[8] These configurations must ensure the constraints (cardinalities, requires/excludes) defined in the DSPL model and include features suitable to the reasoning model.

The choice of the four metamodels, which type the interaction between the components, is open. We currently use SCA to design architectures and two ad hoc metamodels for the DSPL and contexts. We previously used ECA models to specify the adaptation logic;[5] we are now using a goal-based optimization model. Once a designer has chosen the four metamodels, they strongly type the architecture.

Leveraging the models

The sequence diagrams shown in Figure 2 specify the interactions between the architectural components. The key idea is to maintain a context model and an architectural model that both synchronize with the runtime system.

The context model is updated when relevant changes appear in the running system's execution context—for example, CPU load, free memory, or bandwidth. This model is not causally connected with the runtime system—it reflects what happens at runtime (in the execution context), but it should not be directly modified to adapt the running system. On the contrary, the context model serves as a basis for reasoning about the environment and determining a new configuration more adapted to the current context, if necessary.

The architectural model is updated when the running system evolves—that is, when components and bindings are added or removed. Note that this model is not directly manipulated to adapt the running system. On the contrary, the aspect model weaver component produces another architectural model (configuration) when the system should

adapt, depending on the current context model. This configuration is then checked, and the causal connection layer finally realizes the dynamic adaptation,[8] without having to write low-level and error-prone reconfiguration scripts. If the new configuration is not valid, the aspect model weaver component simply discards it and does not proceed to the causal connection layer. Indeed, since the running system has not yet been adapted, it is not necessary to perform a rollback.

Components

Each of the five architectural components has a clear role and well-defined interactions with other components.

Complex event processor. This component observes runtime events generated by probes integrated into the system. When a sequence of events matches a query, expressed in the Event Query Language (EQL), it notifies an observer. This observer knows exactly which model element of the context model it has to update. When this model element is updated, it notifies the goal-based reasoning component. Complex event processing components, such as Esper (http://esper.codehaus.org), allow defining advanced queries on runtime events with time windows and aggregation functions—min, max, average, and so on. Unlike hard thresholds, these queries simplify dealing with permanent context oscillations, for example, by defining thresholds on average values computed on a time slot.

The DCRM's objective is to provide accurate client-related information depending on the context.

Goal-based reasoning engine. When the context model is updated, this component computes a derived DSPL that only contains the mandatory features and a selection of variable features, adapted to the current context. Although we use a goal-based reasoning model, other reasoning models are available. This component is initialized with a DSPL model and a reasoning model. At any time, it is possible to update the DSPL or the reasoning model: add, remove, or update features or reasoning rules, and so on.

Aspect model weaver. This component receives a derived DSPL from the reasoning engine. For all the features of this DSPL, the weaver composes the corresponding aspect to produce a global configuration. This configuration is then checked before it is submitted, if valid, to the configuration manager.

Configuration checker. This online component checks that aspect weaving obtains a consistent configuration.[8] It checks general invariants, which should be enforced in any application, as well as user-defined invariants, which depend on a given application. If the configuration is valid, the aspect model weaver then sends the configuration to the configuration manager.

Configuration manager. This component receives a configuration from the aspect model weaver. It is responsible for configuring and reconfiguring the business architecture. To create and bind components, the configuration manager calls the services offered by the factories (component types).[8] This component maintains a model representing the running system by using the introspection and observation mechanisms provided by the platform. When a new configuration is produced, it compares this new configuration (model) with the current model and deduces a safe sequence of reconfiguration commands. These commands consist of adding or removing components or binding.

DYNAMIC ADAPTATION IN ACTION

We illustrate our approach using the DCRM system developed by CAS Software, our industrial partner in the DiVA project. A simplified version of the system is freely available at www.ict-diva.eu/DiVA/results/tools-and-prototypes/demo.zip/view.

The DCRM's objective is to provide accurate client-related information depending on the context. For example, when the user is working in his office, the system can notify him by e-mail, via a rich Web-based client. He can also access critical resources because he is connected to a trusted network. When the user is driving his car to visit a client, messages received by a mobile or smart phone should notify him of information that is either critical or related to the client. If he is using a mobile phone, he can be notified via the short message service or audio/voice, and Java Telephony API forwards phone calls from his office. If he is using a smart phone, he can also use a lightweight Web client.

Figure 3a shows two reconfiguration scripts generated on the fly. Figure 3b shows the system interface, which consists of two parts:

- *Monitoring.* In the left part of the interface (green rectangle), the user can simulate different environmental variables. Actual runtime sensors are replaced by a check box to simulate Boolean or enumeration (for example, the access point) or sliders to simulate continuous values (for example, the bandwidth). When these elements are activated, they generate exactly the same kind of events as the real sensors. The user thus simulates the environment in a transparent way for the complex event processing component.
- *Reasoning.* The right part of the interface (orange rectangle) graphically represents the DSPL model. Three main features comprise the system: notification, user

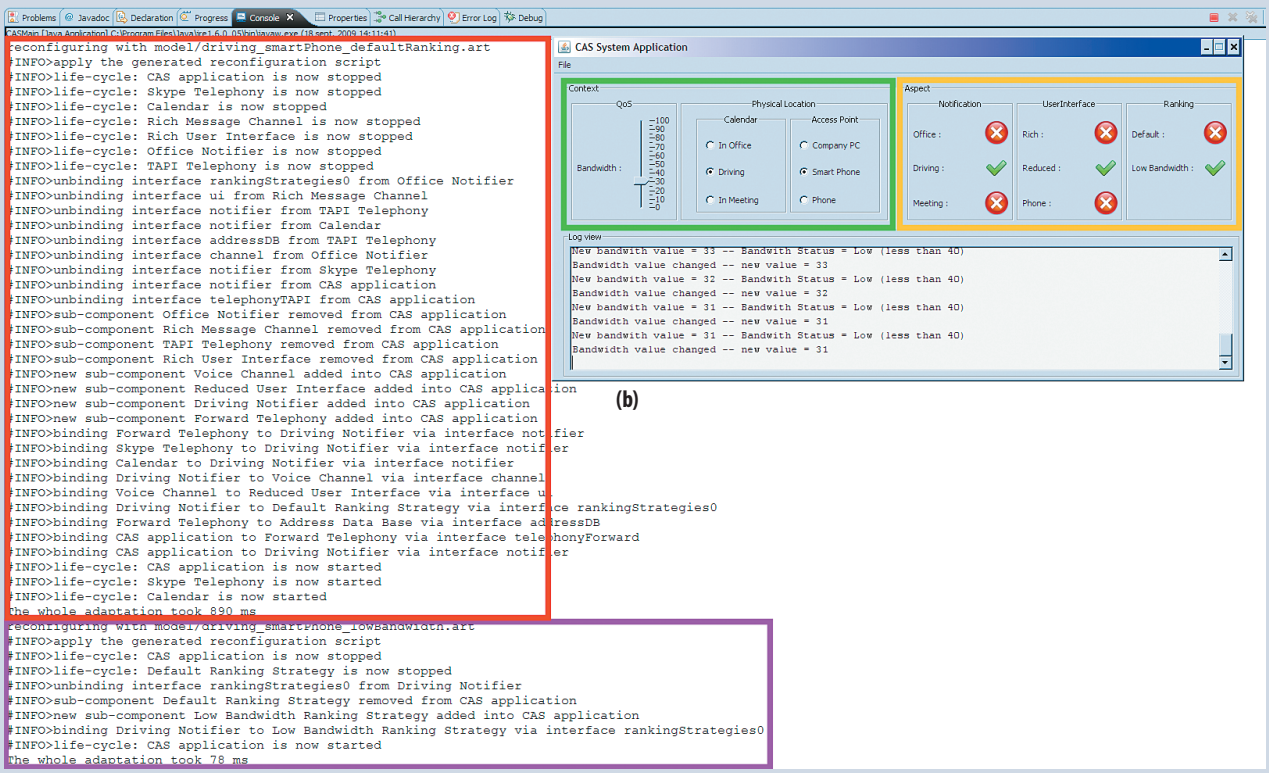


Figure 3. Using the DiVA architecture to manage CAS Software's DCRM system: (a) two reconfiguration scripts generated on the fly; (b) the interface's monitoring and reasoning components.

interface, and ranking. For each feature, the system defines several subfeatures. For example, the system can adapt to provide different notification mechanisms: office, driving, or meeting. An aspect model refines each of these subfeatures. The current configuration, determined by the goal-based reasoning component, is displayed in green.

In the initial context, the user is working in his office. His electronic calendar notifies him that he must visit a client in 30 minutes. When the user logs off his PC and logs on to his smart phone, the reasoning engine computes a new configuration corresponding to a mobile environment. The system replaces the office notifier by a driving notifier, and the user interface switches from rich to reduced. Weaving the associated aspect produces a new configuration. After validating this new configuration, the system automatically generates a reconfiguration script and executes it at runtime, as shown in the top part of Figure 3a (red rectangle).

We assume that the user encounters bandwidth limitation. As the monitoring sequence diagram (Figure 2) shows, when the value exceeds 40 percent of the maximum bandwidth value for 10 seconds, the system updates the context model with bandwidth = high. As soon as the value is below 40 percent, it updates the context model with bandwidth = low. Note that if the bandwidth's value oscillates around 40 percent, the context model will remain stable—bandwidth is low. In Figure 3, this value is currently 32 percent. The reasoning component thus decides to switch from the default ranking strategies to the low bandwidth strategy. Similar to the previous reconfiguration, this generates a script, as illustrated in the bottom part of Figure 3a (purple rectangle).

RELATED WORK

Several other research approaches use architectural models to support dynamic adaptation and software evolution. A decade ago, Peyman Oreizy and colleagues[6] promoted an architecture-based approach to self-adaptive software systems. They stressed that an adaptive system should be open to introducing new behaviors and adaptation plans at runtime.

David Garlan and colleagues[13] also used architectural models for system monitoring and reflection. Specifically, they monitored the executing system to translate observed events to events that construct and update an architectural model that reflects the actual running system. They found that detected inconsistencies could be used to effect runtime adaptations to correct certain type of faults. Simi-

lar to our work, they sought to compare the dynamically determined model with the correct architectural model.

In the context of mobile applications, Jacqueline Floch and colleagues[14] used architectural models and utility functions to describe the dynamic variability of such applications. Their system's main adaptation mechanism replaces the implementation of components at runtime. For each possible implementation (variant) of a component, a fine-grained utility function specifies a precise context in which the variant is useful. Depending on the context, the system integrates most useful variants into the architecture. However, it does not provide support to simulate or validate the adaptation logic at design time.

Our approach focuses on taming the explosion in the number of artifacts while providing a high degree of automation and validation.

The Genie approach[3] also uses architectural models to support the generation and execution of adaptive systems leveraging component-based middleware technologies. A state machine specifies the system's adaptive logic. Each state represents a system configuration, and each transition describes when and how—via reconfiguration scripts—to dynamically switch from one configuration to another. From these models, Genie generates various artifacts such as configuration files and ECA adaptation policies. These artifacts can be dynamically inserted during execution.

Our approach goes one step further than previous efforts. We explicitly design four fundamental aspects of a DAS: its variability (using a feature diagram), the system's environment and the context (valuation of the environment), the adaptation logic, and the system architecture. We particularly focus on taming the explosion in the number of artifacts while providing a high degree of automation and validation. We use AOM techniques to automatically build architectures by composing aspects associated with features, instead of fully specifying all the possible configurations. After validation, we then use MDE techniques to produce reconfiguration scripts that make the system switch from its current configuration to a target configuration more adapted to the current context.

Dynamically adaptive systems play an increasingly vital role in today's society. In addition to CAS Software's DCRM system, we have applied our process to a house-automation system currently deployed in the Rennes metropolitan area in Brittany, France, to help elderly or disabled people remain at home.[8] In this DAS, dynamic adaptation is mostly driven by humans and depends on such factors as the evolution of physical handicaps and the installation of new devices. The number of possible configurations (10^{14}) and transitions (10^{28}) in this system literally explodes. In the context of the DiVA project, we will also apply our approach to an airport crisis management system that should adapt to different crisis types and deal with different roles—for example, airport staff, firemen, and medical staff.

Our tool-supported approach relies on a clear and modular architecture in which components exchange models related to the system's variability, environment, and architecture, and variability is dynamically bound to the context. We do not impose specific metamodels to describe these models. In the DiVA project, we have reused some existing metamodels and designed other ones, and we have used the OSGi platform to implement this architecture.

At design time, engineers can avoid designing by hand all of the system's possible configurations and transitions by explicitly defining a DAS as a DSPL. At runtime, the system analyzes the context and explicitly constructs a suitable configuration using AOM techniques; it validates this configuration using traditional MDE techniques: invariant checking, simulation, and so on. Finally, the system automatically generates a safe reconfiguration script to actually adapt the running business system. If the produced configuration is not consistent, the system simply discards the configuration and derives a new one. Since the running business system has not been adapted yet, it is not necessary to perform a rollback. This process is open to evolution—designers can make the DSPL evolve by seamlessly adding or removing variants, constraints, rules, and so on.

In future work, we plan to improve our reasoning framework and the dynamic adaptation process. In a critical context, the system must react quickly—it can, for example, choose a predefined, prevalidated configuration. In a noncritical context, the system can spend some time to reason and build a suitable configuration. We will investigate the use of bacteriological algorithms to implement reasoning algorithms that can find solutions in a given time budget. Currently, the dynamic adaptation process relies on our aspect model weaver implemented in Kermeta. To make aspect model weaving an efficient solution for dynamic adaptation, we will rely on the compilation feature offered by the language's new version. C

Acknowledgments

This work was partially funded by the DiVA project (EU FP7 STREP, contract 215412, www.ict-diva.eu). The case study presented in this article was provided by CAS Software AG (DiVA industrial partner, www.cas.de/english).

References

1. S. Hallsteinsen et al., "Dynamic Software Product Lines," *Computer*, Apr. 2008, pp. 93-95.
2. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*, 3rd ed., Addison-Wesley Professional, 2001.
3. N. Bencomo et al., "Genie: Supporting the Model-Driven Development of Reflective, Component-Based Adaptive Systems," *Proc. 30th Int'l Conf. Software Eng.* (ICSE 08), ACM Press, 2008, pp. 811-814.
4. J. Zhang and B.H.C. Cheng, "Model-Based Development of Dynamically Adaptive Software," *Proc. 28th Int'l Conf. Software Eng.* (ICSE 06), ACM Press, 2006, pp. 371-380.
5. F. Fleurey et al., "Modeling and Validating Dynamic Adaptation," *Models in Software Eng.*, LNCS 5421, Springer, 2008, pp. 97-108.
6. P. Oreizy et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, May 1999, pp. 54-62.
7. F. Fleurey and A. Solberg, "A Domain-Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems," to appear in *Proc. ACM/IEEE 12th Int'l Conf. Model-Driven Eng. Languages and Systems* (MoDELS 09), ACM Press, 2009.
8. B. Morin et al., "Taming Dynamically Adaptive Systems with Models and Aspects," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), IEEE CS Press, 2009, pp. 122-132.
9. P.C. David and T. Ledoux, "An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components," *Software Composition*, LNCS 4089, Springer, 2006, pp. 82-97.
10. H.J. Goldsby et al., "Goal-Based Modeling of Dynamically Adaptive System Requirements," *Proc. 15th Ann. IEEE Int'l Conf. and Workshop Eng. of Computer Based Systems* (ECBS 08), IEEE CS Press, 2008, pp. 36-45.
11. B. Morin, O. Barais, and J.-M. Jézéquel, "K@rt: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines," *Proc. 3rd Int'l Workshop Models@run.time* (MoDELS 08); www.irisa.fr/triskell/publis/2008/Morin08e.pdf.
12. P.A. Muller, F. Fleurey, and J.-M. Jézéquel, "Weaving Executability into Object-Oriented Meta-Languages," *Proc. MODELS/UML 2005,* LNCS 3713, Springer, 2005, pp. 264-278.
13. D. Garlan et al., "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *Computer*, Oct. 2004, pp. 46-54.
14. J. Floch et al., "Using Architecture Models for Runtime Adaptability," *IEEE Software*, Mar. 2006, pp. 62-70.

Brice Morin *is a PhD student on the INRIA Triskell research team. His research interests include applying model-driven and aspect-oriented techniques to tame the complexity of dynamically adaptive systems, from design time to runtime. He received an MS in computer science from the University of Rennes. Contact him at brice.morin@inria.fr.*

Olivier Barais *is an associate professor at the University of Rennes and member of the INRIA Triskell research team. His research interests include model-driven software engineering, component-based software engineering, and aspect-oriented modeling. He received a PhD in computer science from the University of Lille. Contact him at barais@irisa.fr.*

Jean-Marc Jézéquel *is a professor at the University of Rennes and leader of the INRIA Triskell research team. His research interests include model-driven software engineering for telecommunications and distributed systems. He received a PhD in computer science from the University of Rennes. Contact him at jean-marc.jezequel@irisa.fr.*

Franck Fleurey *is a research scientist in the Model-Driven Software Development Group at SINTEF ICT. His research interests include model-driven engineering, domain-specific languages, variability and adaptation modeling, and model composition. He received a PhD in computer science from the University of Rennes. Contact him at franck.fleurey@sintef.no.*

Arnor Solberg *is a senior research scientist and leader of the Model-Driven Software Development Group at SINTEF ICT. He currently serves as technical manager for the EU-ICT DiVA project. Solberg is an expert on software architectures and software engineering practices. He received a PhD in computer science from the University of Oslo. Contact him at arnor.solberg@sintef.no.*

Selected CS articles and columns are available for free at http://ComputingNow.computer.org

Reach Higher

Advancing in the IEEE Computer Society can elevate your standing in the profession.

- Application in Senior-grade membership recognizes ten years or more of professional expertise.
- Nomination to Fellow-grade membership recognizes exemplary accomplishments in computer engineering.

GIVE YOUR CAREER A BOOST

UPGRADE YOUR MEMBERSHIP

www.computer.org/join/grades.htm

Softw Syst Model
DOI 10.1007/s10270-010-0181-9

THEME SECTION

Reusable model transformations

Sagar Sen · Naouel Moha · Vincent Mahé · Olivier Barais · Benoit Baudry · Jean-Marc Jézéquel

Received: 1 November 2009 / Revised: 29 September 2010 / Accepted: 4 October 2010
© Springer-Verlag 2010

Abstract Model transformations written for an input metamodel may often apply to other metamodels that share similar concepts. For example, a transformation written to refactor Java models can be applicable to refactoring UML class diagrams as both languages share concepts such as classes, methods, attributes, and inheritance. Deriving motivation from this example, we present an approach to make model transformations *reusable* such that they function correctly across several similar metamodels. Our approach relies on these principal steps: (1) We analyze a transformation to obtain an effective subset of used concepts. We prune the input metamodel of the transformation to obtain an effective input metamodel containing the effective subset. The effective input metamodel represents the true input domain of transformation. (2) We adapt a target input metamodel by weaving it with aspects such as properties derived from the effective input metamodel. This adaptation makes the target metamodel a *subtype* of the effective input metamodel. The subtype property ensures that the transformation can process models conforming to the target input metamodel without *any change* in the transformation itself. We validate our approach by adapting well known refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method) written for an in-house domain-specific modeling language (DSML) to three different industry standard metamodels (Java, MOF, and UML).

Keywords Adaptation · Aspect weaving · Genericity · Metamodel pruning · Model typing · Model transformation · Refactoring

1 Introduction

Model transformations are software artifacts that underpin complex software system development in Model-driven Engineering (MDE). Making model transformations *reusable* is the subject of this paper.

Software reuse in general has been largely investigated in the last two decades by the software engineering community [1,2]. Basili et al. [3] demonstrate the benefits of software reuse on the productivity and quality in object-oriented systems. However, reuse is a new entrant in the MDE community [4]. One of the primary difficulties in making a model transformation reusable across different input domains is the difference in structural aspects between commutable/interchangeable input metamodels. Consider an example where model transformation reuse becomes obvious and yet is infeasible due to structural differences in commutable input metamodels. The example consists of a model transformation to *refactor models of class diagrams*, which is possible in

Communicated by Tony Clark and Jorn Bettin.

S. Sen (✉) · N. Moha · V. Mahé · O. Barais · B. Baudry · J.-M. Jézéquel
INRIA Rennes, Bretagne Atlantique/IRISA, Université Rennes 1, Triskell Team, Campus de Beaulieu, 35042 Rennes Cedex, France
e-mail: ssen@irisa.fr

N. Moha
e-mail: moha@irisa.fr

V. Mahé
e-mail: vmahe@irisa.fr

O. Barais
e-mail: barais@irisa.fr

B. Baudry
e-mail: bbaudry@irisa.fr

J.-M. Jézéquel
e-mail: jezequel@irisa.fr

Published online: 11 November 2010

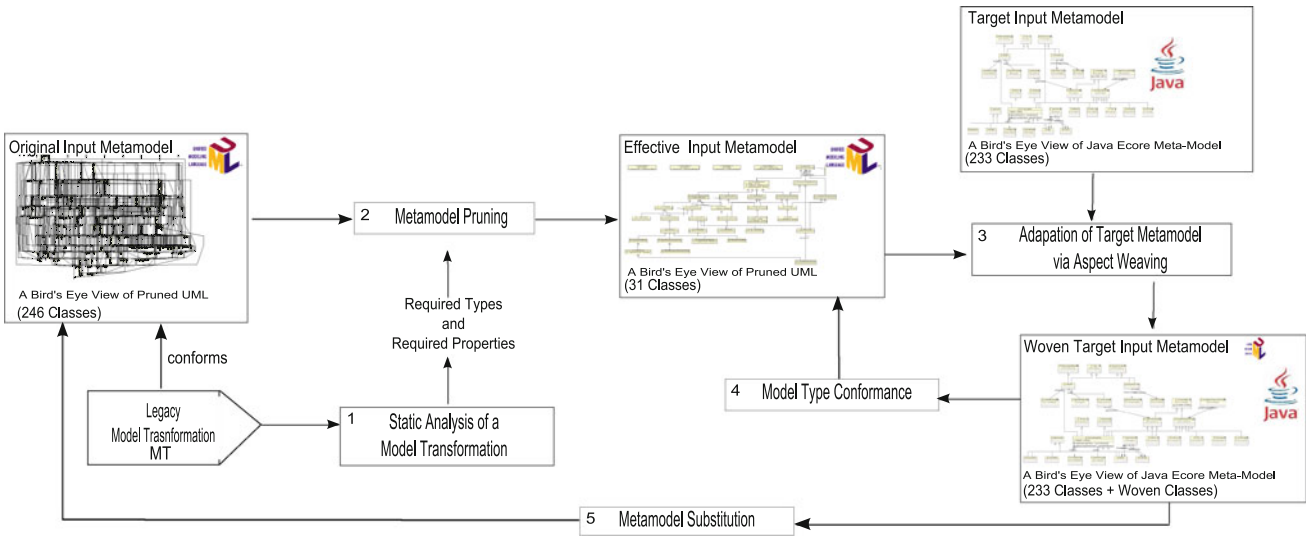


Fig. 1 Overview of the approach

several modeling languages supporting the concepts/types of classes, methods, attributes, and inheritance. For instance, the metamodels for the languages Java, MOF (Meta Object Facility), and UML all contain concepts/types needed to specify classes. If we emphasize the necessity for reuse then the refactoring transformation must be intuitively adaptable to all three metamodels as they manipulate similar models containing objects of similar types. Hence, we ask: How do we reuse one implementation of a model transformation for other type-theoretically similar modeling languages?

Our aim is to enable flexible reuse of model transformations across various type-theoretically similar metamodels to enhance productivity and modularity in MDE. The aim is in line with the effort of the model-driven engineering community to provide methods and techniques for the development of tool chains through model-driven interoperability. In this paper, we present an approach to make legacy model transformations reusable for different target input metamodels. We do not touch the body of the legacy transformation itself but transform a target input metamodel such that it becomes a *subtype* of the effective subset of the transformation's input metamodel. We call the effective subset an *effective input metamodel* which represents the true input domain of the legacy model transformation. By definition in model type theory [5] the subtype property or the type conformance permits the legacy model transformation to process pertinent models conforming to the target input metamodel. Concisely, our approach, depicted in Fig. 1, follows these steps: (1) We perform a static analysis of the legacy transformation to extract the types and properties required in the transformation. (2) We automatically obtain an effective input metamodel via *metamodel pruning* [6] based on the types and properties required in the transformation. Metamodel pruning [6] is an algorithm that conserves a set of required types and properties (given as input to metamodel pruning) and all its obligatory dependencies while it prunes away every other type and property. Hence, it outputs an effective subset metamodel of a possible large input metamodel such as the UML. This step drastically reduces the adaptation effort in the next step when dealing with large metamodels such as UML where model transformations often use only a small subset of the entire metamodel. (3) We adapt a target input metamodel by weaving it with structural aspects from the effective input metamodel. We also weave accessor functions for these structural aspects that seek information from related types in the target input metamodel. For example, if Java is the target input metamodel and UML is the original input metamodel for a legacy transformation then values from properties in Java input models must stay synchronized with UML properties actually handled by the transformation. Moreover, the Java input models must temporarily (during execution) contain properties derived from UML that are identifiable by the transformation for UML models. These identifiable properties in fact are part of the effective input metamodel. Therefore, the woven aspects are derived properties and accessor functions that help make the Java metamodel a subtype of UML. (4) We use *model typing* [7] to verify the type conformance between the woven target input metamodel and the effective input metamodel. The woven target input metamodel must be a *subtype* of the effective input metamodel. Our approach is infeasible when the target input metamodel cannot be adapted to show type conformance with the effective input metamodel for some type(s) used in the transformation. (5) Replacing the original input metamodel with the woven target input metamodel at execution allows the legacy model transformation to process relevant input models conforming to the target input metamodel.

The scientific contribution of our approach is based on a combination of two recent ideas; namely, metamodel pruning [6] and manual specification of generic model refactorings [8]. In [8], the authors manually specify a generic model transformation for a hand-made generic metamodel that is adapted to various target input metamodels. The generic metamodel, presented in [8], is a lightweight metamodel that contains a minimum set of concepts (such as classes, methods, attributes, and parameters) common to most metamodels for object-oriented design/development such as Java and UML. In our work, we automatically synthesize an effective input metamodel via metamodel pruning [6], which is in contrast to manually specifying a generic metamodel as done in [8]. Further, the effective input metamodel is derived from an arbitrary input metamodel of a legacy model transformation and not from a domain-specific generic metamodel (for refactoring) as in [8]. The adaptation of target input metamodels to the effective input metamodel via aspect weaving remains similar to the approach in [8].

We demonstrate our approach on well known model transformations; namely, refactorings [9]. A refactoring is a particular transformation performed on the structure of software to make it easier to understand and modify without changing its observable behavior [9]. For example, the refactoring Pull Up Method consists of moving methods to the superclass if these methods have the same signatures and/or results on subclasses [9]. We validate our approach by performing some experiments where three well known legacy refactorings (Encapsulate Field, Move Method, and Pull Up Method) are adapted to three different industrial metamodels (Java, MOF, and UML). The legacy refactorings are written in the transformation language Kermeta [10].

This article is organized as follows. In Sect. 2, we describe motivating examples that illustrate the key challenges. In Sect. 3, we introduce foundations necessary to describe our approach. The foundations include a description of the executable metamodeling language, Kermeta, highlighting some of its new features including the notion of model typing, and a presentation of meta-model pruning to obtain an effective input metamodel. Section 4 gives a general step-by-step overview of our approach. Section 5 describes the experiments that we performed for adapting three legacy refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method) initially described for an in-house DSML to three different industry standard metamodels (Java, MOF, and UML). Section 6 surveys related work. Section 8 concludes and presents future work.

2 Motivating examples

Let us suppose that a company needs to economically upgrade its legacy transformations from an old input metamodel to a new but similar industry standard metamodel such as the latest UML. The old metamodel may either be from an in-house DSML or an old version of an industry standard such as UML. The legacy transformation itself must remain unchanged.

Let us now consider an example of a model transformation that can refactor the in-house DSML. The DSML itself is used to model software structure and behaviour. Our ultimate objective is to make this model transformation reusable and applicable across different industry standard metamodels. Specifically, we describe the Pull Up Method refactoring transformation which we intend to use for models from three different metamodels (Java, MOF, and UML).

2.1 The Pull Up Method refactoring

The Pull Up Method refactoring consists of moving methods to the superclass when methods with identical signatures and/or results are located in sibling subclasses [9]. This refactoring aims to eliminate duplicate methods by centralizing common behavior in the superclass. A set of preconditions must be checked before applying the refactoring. For example, one of the preconditions to be checked consists of verifying that the method to be pulled up is not a constructor. Another precondition checks that the method does not override a method of the superclass with the same signature. A third precondition consists of verifying that methods in sibling subclasses have the same signatures and/or results.

The example of the Pull Up Method refactoring presented in [11] of a Local Area Network (LAN) application [12] and adapted in Fig. 2 shows that the method `bill` located in the classes `PrintServer` and `Workstation` is pulled up to their superclass `Node`.

The Pull Up Method refactoring is written for an in-house DSML for the INRIA team TRISKELL from Rennes, France that contains the notions of classes, attributes, inheritance, methods and several other concepts related to contracts and verification that are not pertinent to refactoring. The in-house DSML does not conform to an industry standard metamodel such as UML.

2.2 Three different metamodels

Our goal is to make the refactoring reusable across three different target input metamodels (Java, MOF, and UML), which support the definition of object-oriented structures (classes, methods, attributes, and inheritance). The Java metamodel described in [13] represents Java programs with some restrictions over the Java code. For example, inner classes, anonymous classes, and generic types are not modeled. As a MOF metamodel, we consider the metamodel of Kermeta [10], which is an extension of MOF [14] with an imperative action language for specifying constraints and

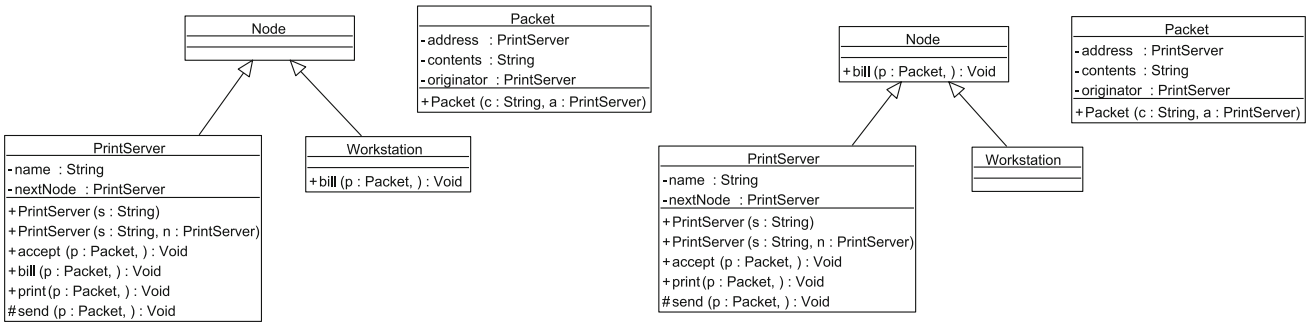


Fig. 2 Class diagrams of the LAN application before and after the Pull Up Method Refactoring of the Method `bill`

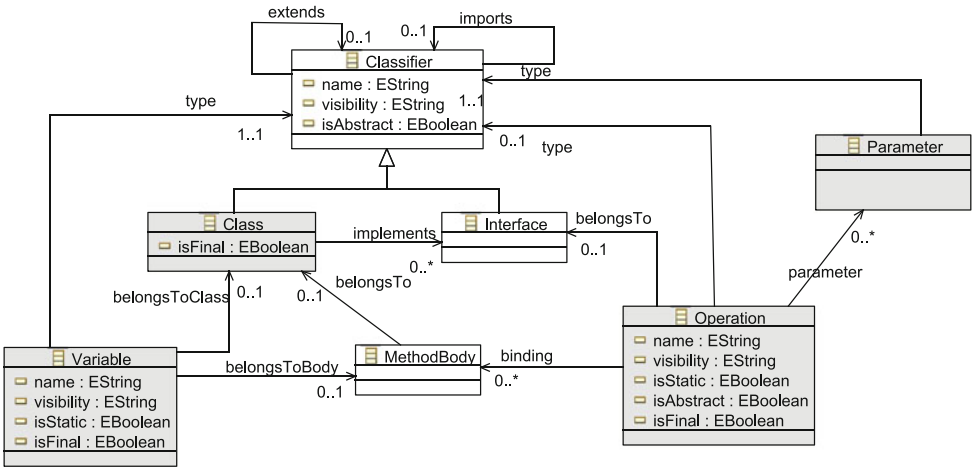


Fig. 3 Subset of the Java Metamodel

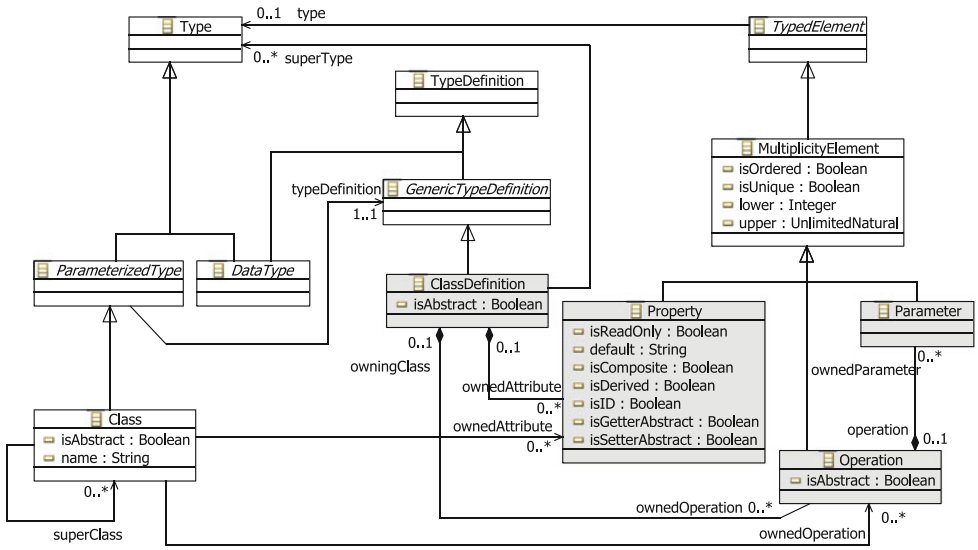
operational semantics of metamodels. The UML metamodel studied in this paper corresponds to version 2.1.2 of the UML specification [15]. This Java metamodel is *one* possible specification for Java programs; we may use another Java metamodel based on the specification of the Abstract Syntax Tree Metamodel (ASTM) provided by the OMG ADM (Architecture-Driven Modernization) group [16].

We provide an excerpt of each of these metamodels in Figs. 3, 4, and 5. These metamodels share some commonalities, such as the concepts of classes, methods, attributes, parameters, and inheritance (highlighted in grey in the figures). These concepts are necessary for the specification of refactorings, and in particular for the Pull Up Method refactoring. However, they are represented differently from one metamodel to another as detailed in the next paragraph.

2.3 Problems

We encounter several problems if we intend to specify a common Pull Up Method refactoring for all three metamodels:

- **The metamodel elements** (such as classes, methods, attributes, and references) **may have different names**. For example, the concept of attribute is named `Property` in the MOF and UML metamodels whereas in the Java metamodel, it is named `Variable`.
- **The types of elements may be different**. For example, in the UML metamodel, the attribute `visibility` of `Operation` is an enumeration of type `VisibilityKind` whereas the same attribute in the Java metamodel is of type `String`.
- There may be **additional or missing elements** in a given metamodel compared to another. For example, `Class` in the UML metamodel and `ClassDefinition` in the MOF metamodel have several superclasses whereas `Class` in the Java metamodel has only one. Another example is the `ClassDefinition` in MOF, which is missing an attribute `visibility` compared to the UML and Java metamodels.
- **Opposites may be missing in relationships**. For example, the opposite of the reference related to the notion of inheritance (namely, `superClass` in the MOF and UML metamodels, and `extends` in the Java metamodel) is missing in the three metamodels.
- **The way metamodel classes are linked together may be different** from one metamodel to another. For example, the classes `Operation` and `Variable` in the Java metamodel are not directly accessible from `Class` as



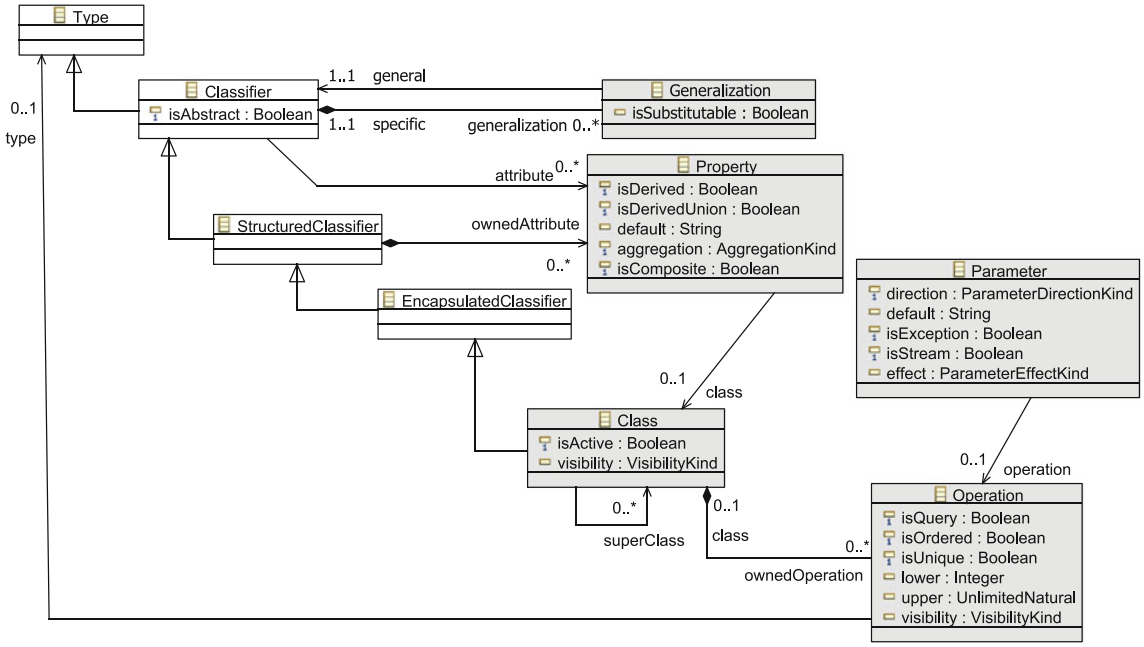


Fig. 4 Subset of the MOF Metamodel

Fig. 5 Subset of the UML Metamodel

opposed to the corresponding classes in the MOF and UML metamodels.

These differences among these three metamodels make it impossible to directly reuse a Pull Up Method refactoring across all three metamodels. Hence, we are forced to write three different implementations of the same refactoring transformation for each of the three metamodels. We address this problem with our approach in Sect. 4. In the approach we make a single transformation reusable across different metamodels without rewriting the transformation. We only adapt different target input metamodels such that they become a subtype of the input metamodel of the transformation.

3 Foundations

This section presents the foundations required to explain the approach presented in Sect. 4. We describe the model transformation language Kermeta in Sect. 3.1. We present relevant Kermeta features that allow weaving aspects into target input metamodels in Sect. 3.2. We describe Kermeta's implementation of model typing in Sect. 3.3 which helps us perform

all type conformance operations in our approach. Finally, in Sect. 3.4 we present the metamodel pruning algorithm to obtain the effective input metamodel to be used in the approach.

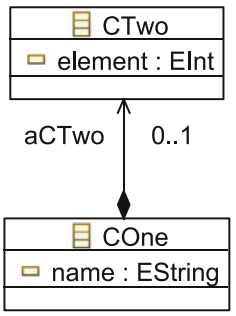
3.1 Kermeta

Kermeta is a language for specifying metamodels, models, and model transformations that are compliant to the MOF standard [14]. The object-oriented meta-language MOF supports the definition of metamodels in terms of object-oriented structures (packages, classes, attributes, and methods). It also provides model-specific constructions such as containments and associations between classes. Kermeta extends the MOF with an *imperative action language* for specifying constraints and operational semantics for metamodels [17]. Kermeta is built on top of Eclipse Modeling Framework (EMF) within the ECLIPSE development environment. The action language of Kermeta provides mechanisms for dynamic binding, reflection, and exception handling. It also includes classical control structures such as blocks, conditionals, and loops. We note that Kermeta is used to specify the refactorings used in our examples in Sect. 5.

3.2 Features of Kermeta

The action language of Kermeta provides some features for weaving aspects, adding derived properties, and specifying constraints such as invariants and pre-/post-conditions. Indeed, the first feature of Kermeta is its ability to extend an existing metamodel with new structural elements (classes, methods, and attributes) by weaving aspects (similar to inter-type declarations in AspectJ or open-classes [18]).Aspect weaving consists of composing a base model with aspects defining new concerns, thereby yielding a base model with new structure and behavior. This feature offers more flexibility to developers by enabling them to easily manipulate and reuse existing metamodels while separating concerns. The second key feature is the possibility to add derived properties. A derived property is a property that is derived or computed through *getter* and *setter* accessors for simple types and `add` and `remove` methods for collection types. The derived property thus contains a body, as operations do, and can be accessed in read/write mode. The feature amounts to the possibility of determining the value of a property based on the values of other properties. These other properties may come from the same class and/or from properties reachable through the navigation of the metamodel. The last pertinent Kermeta feature is the specification of pre- and post-conditions on operations and invariants on classes. These assertions can be directly expressed in Kermeta or imported from OCL (Object Constraint Language) files [19].

Fig. 6 Metamodel *M*



3.3 Model typing

The Kermeta language integrates the notion of model typing [7], which corresponds to a simple extension to object-oriented typing in a model-oriented context. Model typing can be related to structural typing found in languages such as Scala. Indeed, a model typing is a strategy for typing models as collections of interconnected objects while preserving type conformance, used as a criterion of substitutability.

The notion of model type conformance (or substitutability) has been adapted and extended to model types based on Bruce's notion of type group matching [20]. The matching relation, denoted $<\#$, between two metamodels defines a function of the set of classes they contain according to the following definition:

> Metamodel M' matches another metamodel M (denoted $M' < \#M$) iff for each class C in M, there is one and only one corresponding class or subclass C' in M' such that every property p and operation op in $M.C$ matches in $M'.C'$, respectively, with a property p' and an operation op' with parameters of the same type as in $M.C$.

This definition is adapted from [7] and improved here by relaxing two strong constraints. First, the constraint related to the name-dependent conformance on properties and operations was relaxed by enabling their renaming. The second constraint related to the strict structural conformance was relaxed by extending the matching to subclasses.

Let's illustrate model typing with two metamodels M and M' given in Figs. 6 and 7. These two metamodels have model elements that have different names and the metamodel M' has additional elements compared to the metamodel M.

> $C1 < \#COne$ because for each property $COne.p$ of type D (namely, $COne.name$ and $COne.aCTwo$), there is a matching property $C1.q$ of type D' (namely, $C1.id$ and $C1.aC2$), such that $D' < \#D$.

Thus, $C1 < \#COne$ requires $D' < \#D$, which is true because:

- $COne.name$ and $C1.id$ are both of type $String$.
- $COne.aCTwo$ is of type $CTwo$ and $C1.aC2$ is of type $C2$, so $C1 < \#COne$ requires $C2 < \#CTwo$ or

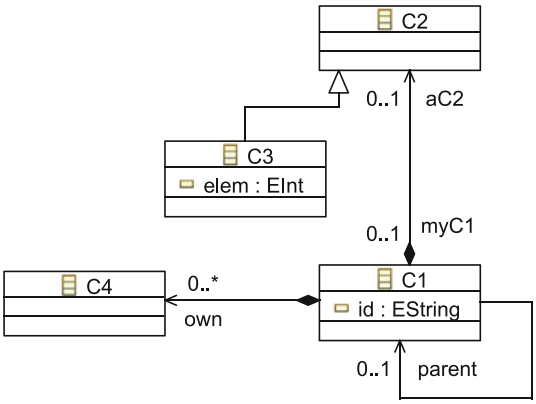


Fig. 7 Metamodel M'

that a subclass of *C2* matches *CTwo*. Only $C3 < \#CTwo$ is true because *CTwo.element* and *C3.elem* are both of type *String*.

Thus, matching between classes may depend on the matching of their related dependent classes. As a consequence, the dependencies involved when evaluating model type matching are heavily cyclical [5]. The interested reader can find in [5] the details of matching rules used for model types.

However, model typing with the mechanisms of renaming and inheritance is not sufficient for matching metamodels that are structurally different. We overcome this limitation of the model typing by weaving required aspects as described in our approach in Sect. 4.

3.4 Metamodel pruning

Metamodel pruning [6] is an algorithm that outputs an effective subset metamodel of a possible large input metamodel such as UML. The output effective metamodel conserves a set of required types and properties (given as input to metamodel pruning) and all its obligatory dependencies (computed by the algorithm). The algorithm prunes every other type and property. In the type-theoretic sense the resulting effective metamodel is a *supertype* of the large input metamodel. We verify the supertype property using model typing [7]. We concisely describe the metamodel pruning algorithm in the following paragraphs.

Given a possibly large metamodel such as UML that may represent the input domain of a model transformation, we ask the question : Does the model transformation process models containing objects of all possible types in the input metamodel? In several cases the answer to this question may be no. For instance, a transformation that refactors UML models only processes objects with types that come from concepts in the UML class diagrams subset but not UML Activity, UML Statechart, or UML Use case diagrams. How do we obtain this effective subset? This is the problem that metamodel pruning solves.

The principle behind pruning is to preserve a set of required types T_{req} and required properties P_{req} and prune away the rest in a metamodel. The authors of [6] present a set of rules that help determine a set of required types T_{req} and required properties P_{req} given a metamodel MM and an initial set of required types and properties. The initial set may come from various sources such as manual specification or a static analysis of model transformations to reveal used types. A rule in the set adds all superclasses of a required class into T_{req}. Similarly, if a class is in T_{req} or is a required class then for each of its properties p, add p into P_{req} if the lower bound for its multiplicity is > 0. Apart from rules, the algorithm contains options which allow better control of the algorithm. For example, if a class is in T_{req} then we add all its subclasses into T_{req}. This optional rule is not obligatory but may be applicable under certain circumstances giving the user some freedom. The rules are executed where the conditions match until no rule can be executed any longer. The algorithm terminates for a finite metamodel because the rules do not remove anything from the sets T_{req} and P_{req}.

Once we compute the sets T_{req} and P_{req} the algorithm simply removes the remaining types and properties to output the effective metamodel MM_e. The effective metamodel MM_e generated using the algorithm in [6] has some very interesting characteristics. Using model typing (discussed in Sect. 3.3) we verify that MM_e is a *supertype* of the metamodel MM. This implies that all operations written for MM_e are valid for the large metamodel MM.

4 Approach

We present an approach to make a legacy model transformation `MT` reusable. We outline the approach in Fig. 1 and describe the steps in the approach below:

Step 1: Static analysis of a transformation

As shown in Fig. 1 we first perform *static analysis* on the legacy model transformation `MT`. Static analysis can be extrapolated to several model transformations when they are called and navigable from a main transformation. The main transformation is given as an input to the static analysis process. The static analysis involves visiting each rule, each constraint, and each statement in the model transformation to obtain an initial set of required types T_{req} and a set of required properties P_{req} manipulated in the input metamodel `Input-MM`. The goal behind performing static analysis is to find the *subset of concepts* in the input metamodel *actually used in the transformation*. We do not go into the details of the static analysis process as it is just classical traversal of the *abstract syntax tree* of an entire program or a rule in order to check the

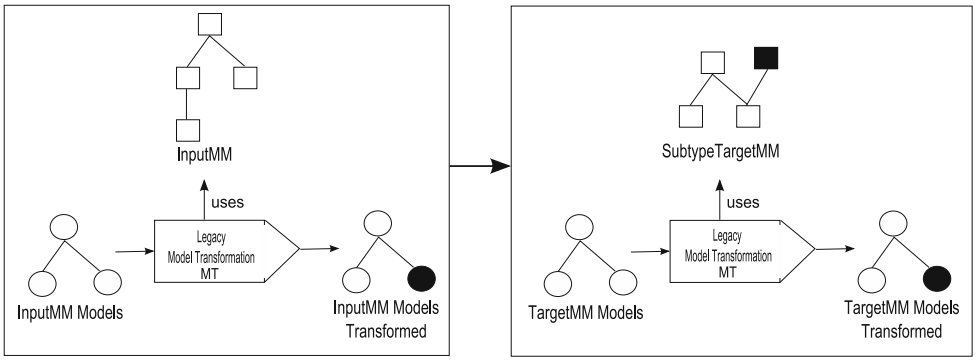


Fig. 8 The Legacy Model Transformation `MT` used as Transformation for Subtype `TargetMM`

type of each term. The static analysis can only be performed when the source code for the transformation is available. If not, the required types and properties must be manually specified. If the type is present in `InputMM` we add it to T_{req}. Similarly, we add all properties manipulated and existing in `InputMM` into P_{req}.

Step 2: Metamodel pruning

Using the set of required types T_{req} and properties P_{req} we perform metamodel pruning on `InputMM` to obtain an effective input metamodel `EffectiveMM` that is a *supertype* of `InputMM`. We recall the metamodel pruning algorithm described in Sect. 3.4. The algorithm generates the minimal effective input metamodel `EffectiveMM` that contains the required types and properties and their obligatory dependencies. The advantages of automatically obtaining the `EffectiveMM` are the following:

- The `EffectiveMM` represents the true input domain of the legacy model transformation `MT`.
- The `EffectiveMM` containing only relevant concepts from the `InputMM` drastically reduces the number of aspect weaving and type matching operations to be performed in Step 3. There is often a combinatorial explosion in the number of type comparisons given that each concept in the input metamodel must be compared with the target metamodel `TargetMM`

The metamodel pruning process plays a key role when the input domain of a transformation corresponds to a standard metamodel such as UML where the number of classes is about 246 and properties about 587. Writing adaptations for each of these classes, as we shall see in Step 3, becomes very tedious unless only a subset of the input metamodel is in use.

Step 3: Aspect weaving of target metamodel

One of the new features of Kermeta is to weave aspects (see Sect. 3.2) into metamodels. In the third step we *manually* identify and weave aspects from `EffectiveMM` into the `TargetMM`. We also weave *getter* and *setter* accessor functions into `TargetMM`. These accessors seek information in related concepts of the `TargetMM` and assigns their values to the initially woven properties and types from `EffectiveMM`. We verify the subtype property as described in Step 4. Examples of woven aspects are given in Sect. 5.

Step 4: Model type conformance

We perform model type conformance between the effective input metamodel `EffectiveMM` and the target input metamodel `TargetMM` with woven properties. The model type matching process is described in Sect. 3.3. All the types in the woven `TargetMM` are matched against each type in `EffectiveMM`. If all types match, then `TargetMM` with aspects is recalled as the subtype target input metamodel: `SubTypeTargetMM`. Replacing the input metamodel of the legacy model transformation `MT` with `SubTypeTargetMM` will allow all pertinent models conforming to the target input metamodel to be processed by `MT`, as shown in Fig. 8.

5 Experiments and discussion

We performed some experiments by applying our approach to legacy model refactoring transformations (Encapsulate Field, Move Method, and Pull Up Method [9]) written for an in-house DSML to three target industry standard input metamodels Java, UML, and MOF. Our goal is to be able to reuse these three well-known refactorings on models of a given application conforming to the three different metamodels. We choose an application for the simulation of a local area network (LAN) developed by the Vrije Universiteit Brussel and the University of Bern [12]. This application has been used to illustrate various aspects of the evolution of object oriented programs and is thus appropriate for the application of refactorings.

We illustrate our approach using the specific example of the `Pull Up Method` refactoring transformation. The implementation of the example is in Listing 1 (see Appendix A.1) which is an excerpt of the class `Refactor`. The class `Refactor` contains the operation `Pull Up Method` (Line 5). The refactoring is implemented in

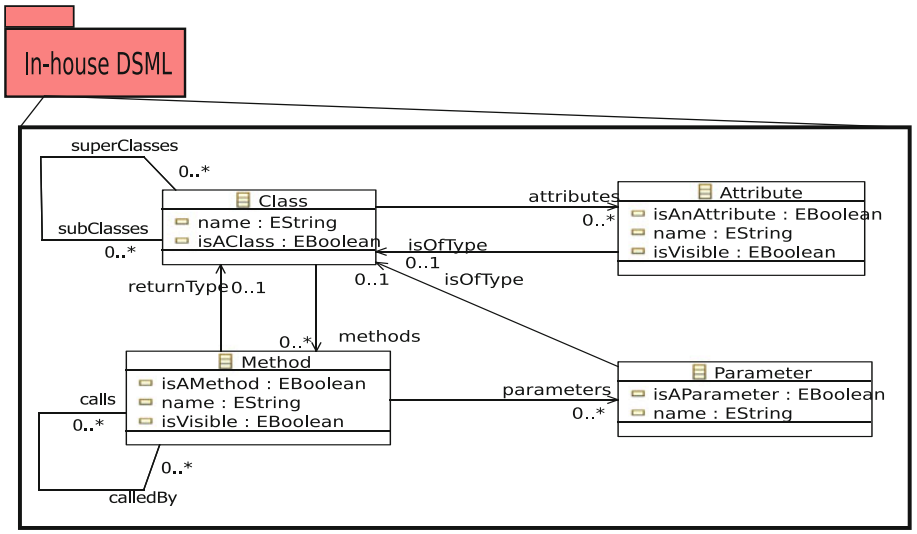


Fig. 9 Effective Metamodel `EffectiveMM` extracted from an In-house DSML via Pruning

Kermeta[1]. This operation aims to pull up the method `meth` from the source class `source` to the target class `target`. This operation contains a precondition that checks if the sibling subclasses have methods with the same signatures. In the body of the operation, the method `meth` is added to the methods of the target class and removed from the methods of the source class.

A step-by-step application of our approach is described in Sect. 5.1. We discuss the experiment in Sect. 5.2.

5.1 Application

In *Step 1*, we perform a static analysis of refactoring model transformations (Encapsulate Field, Move Method, and Pull Up Method) applied on an in-house DSML for the INRIA team TRISKELL from Rennes, France. The result of the static analysis is a set of required types and required properties. The analysis reveals that required classes in the transformation are : `Class`, `Attribute`, `Method`, and `Parameter`. This drastically reduces the number of adaptations required in the target input metamodels: Java, MOF, and UML. The DSML contains several other classes related to contracts and verification. These classes and their properties are not used by the refactoring transformation and hence the static analysis does not reveal them. Due to space limitations we do not show the entire DSML in the paper.

In *Step 2*, we perform metamodel pruning of the input metamodel `InputMM` for the refactoring transformation. We show the resulting effective input metamodel `EffectiveMM` in Fig. 9. As claimed earlier the effective metamodel only contains the required types, required properties, and their obligatory dependencies. The only inputs to the metamodel pruning algorithm were the classes `Class`, `Attribute`, `Method`, and `Parameter`. The rest of the obligatory structure for the `EffectiveMM` metamodel is automatically conserved by the metamodel pruning algorithm. All other irrelevant classes for statecharts, verification, and activities are automatically removed.

In *Step 3*, we adapt the target input metamodels to the effective input metamodel `EffectiveMM` using the Kermeta features for weaving aspects and adding derived properties. We weave missing types, properties and their opposite properties from the `EffectiveMM` into the `TargetMM`. These properties include *getter* and *setter* accessors that seek information in the `TargetMM` to assign values to the derived properties woven from `EffectiveMM`. This step of adaptation is necessary because model typing is too restrictive for allowing a matching between metamodels that are structurally very different. The adaptation virtually modifies the structure of the target input metamodel with additional elements and in the following step we use model typing to match the metamodels. The resulting subtype target input metamodel is $SubtypeTargetMM$, as seen in Fig. 8.

To better understand the adaptation process we illustrate it with a simple example shown in Fig. 10. In Fig. 10 (a), type `Class` exists in the effective input metamodel `EffectiveMM` and `Classifier` exists in the target input metamodel `TargetMM` Java. We ask in Fig. 10 (b) if the the types match with respect to model typing rules and the answer is *no* because the properties `superClasses` and `subClasses` do not appear in `TargetMM`. Hence, we weave the properties `superClasses` and `subClasses` from `Class` into `Classifier` as shown in Fig. 10 (c). These properties are computed using the already existing property `extends` in `TargetMM`. Now, the types `Class` and `Classifier` match as seen in Fig. 10 (d). This process is repeated for *every type* in `EffectiveMM` such that a conforming type is created in the target input metamodel. If a match for a type is not found or multiple matches for a

[1] The interested reader can refer to the Kermeta syntax in [21].

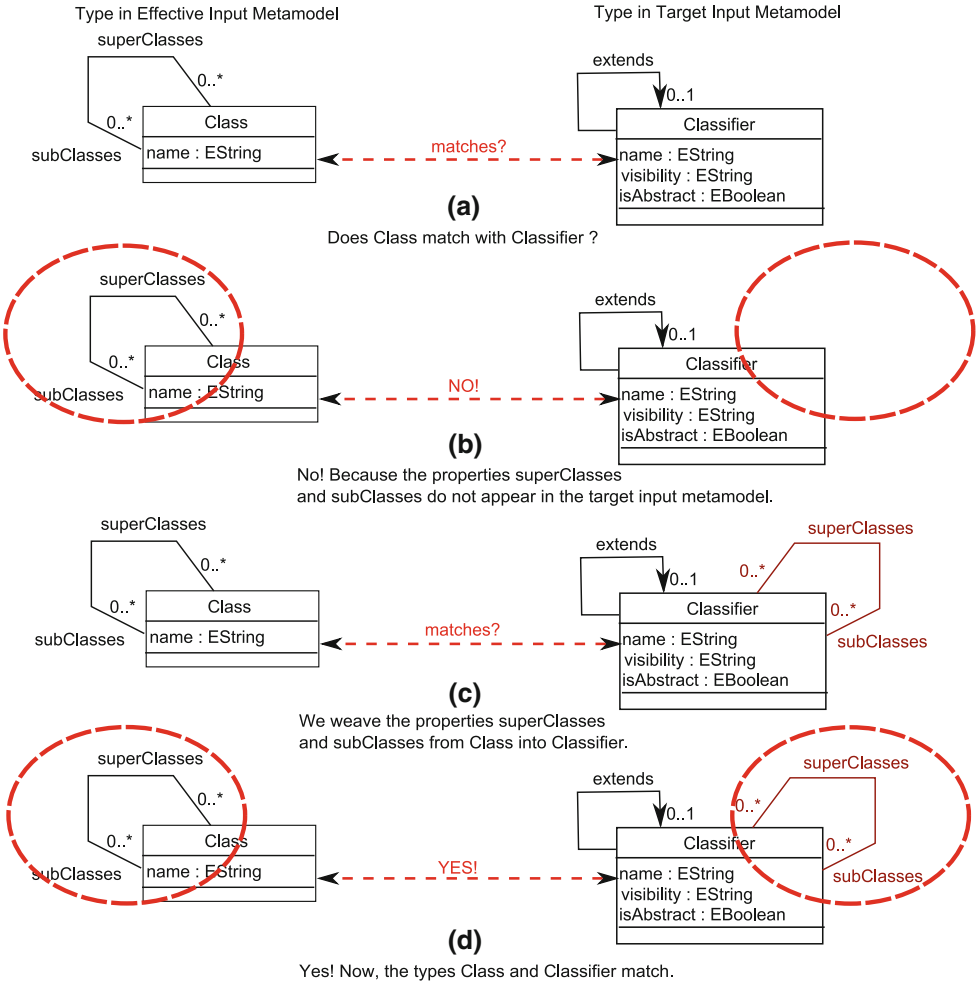


Fig. 10 An example of weaving steps for adaptation

type in the `EffectiveMM` are found then the target input metamodel is unadaptable and our approach fails.

In the following paragraphs, we describe technical details of the adaptations for the target input metamodels Java, MOF, and UML such that they type conform with the effective input metamodel `EffectiveMM` of refactoring transformations. In particular, we describe the adaptations of the derived properties `superClasses` and `subClasses` of `Class` for the target input metamodels. We discuss only the woven getter accessors of the derived properties; the setter accessors are symmetric.

Adaptation for the Java metamodel Listing 2 in Appendix A.2 describes the adaptation made to the Java metamodel to adapt it to `EffectiveMM`. The adaptation is applied on a subset of the Java metamodel shown in Fig. 3. The derived property `superClasses` corresponds to a simple access to the property `extends` that is then wrapped in a Java `Class` (Lines 12–15). For the derived property `subClasses`, the opposite `inv_extends` of the property `extends` was weaved by an aspect on the class `Classifier` and used to get the set of subclasses (Lines 17–21).

Adaptation for the MOF metamodel Listing 3 in Appendix A.3 describes the adaptation made to the MOF metamodel to adapt it to `EffectiveMM`. We apply the adaptation on a subset of the MOF metamodel shown in Fig. 4. Due to the distinction in the MOF between `Type` and `TypeDefinition` to handle the generic types, it is less straightforward to compute the derived properties `superClasses` and `subClasses`. Several opposites are required as shown in Listing 3 (Lines 5–15).

Adaptation for the UML metamodel Listing 4 in Appendix A.4 describes the adaptation made to the UML metamodel to adapt it to `EffectiveMM`. We apply the adaptation on a subset of the UML metamodel shown in Fig. 5. In UML, the inheritance links are reified through the class `Generalization` (Lines 5–7). Thus, the derived property `superClasses` is computed by accessing the class `Generalization` and the reference property `general` (Lines 11–15). As in Java and MOF, an opposite `inv_general` is specified to get the set of subclasses (Lines 17–21).

Finally, we apply the refactoring on the target input metamodels as illustrated in Listing 5 (see Appendix A.5) for the UML metamodel. We reuse the example of the method

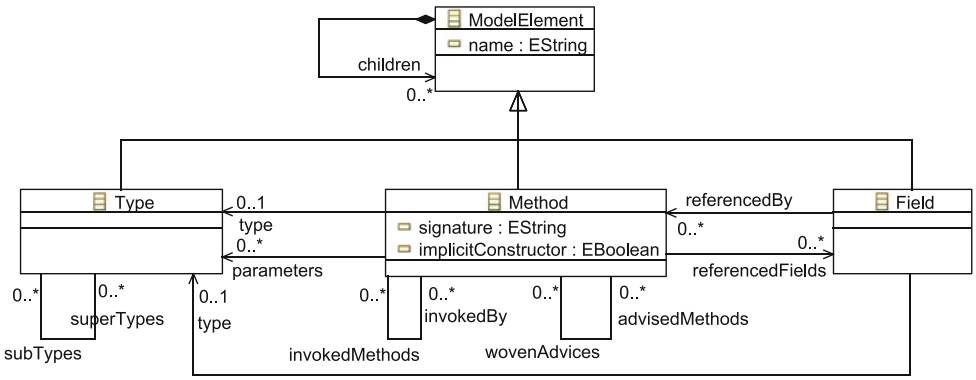


Fig. 11 Subset of the Fourth Metamodel

`bill` in the LAN application (Lines 12, 16). We notice that the class `Refactor` takes as an argument the UML metamodel (Lines 18–19), which due to the adaptation of Listing 4 is now a subtype of the expected supertype `EffectiveMM` as specified in Listing 1. The model typing guarantees the type conformance between the UML metamodel and the effective input metamodel `EffectiveMM`.

5.2 Discussion

We also experimented with a fourth metamodel as shown in Fig. 11. In this metamodel, the two classes (corresponding to `Class` and `Parameter` in the effective input metamodel) are unified in the same class (`Type`). This case introduced an ambiguous matching with the effective input metamodel since these classes are distinct in the latter. This special case illustrates a limitation of our approach that needs to be overcome and will be investigated in future work. Thus, the only prerequisite of our approach is that each element in the effective input metamodel should correspond to a distinct element in the target input metamodel. We specified a fourth refactoring, Extract Method [9], that creates a new method from a code fragment. The Extract Method refactoring uses the concept of *method body*, but this concept is missing in the UML metamodel. Therefore, the missing concept prevents the ability to reuse the refactoring for the UML metamodel. However, during the adaptation step, we could fill in this difference by weaving the missing concept to the UML metamodel as a stub. Our approach is thus not very restrictive since the mechanism of adaptation enables the raising of awareness of inherent limitations.

In [10], we apply some refactorings on a Java metamodel with a flat structure (*i.e.*, with no containers). The search for elements in such a metamodel is not optimal since we need to traverse all elements in the flat structure. However, in the current paper, the navigation of the elements is easier due to opposites properties. These properties enable bi-directional traversal of a metamodel. The addition of opposites is done automatically while loading metamodels in the Kermeta platform.

Our approach theoretically relies on the model typing and is feasible in practice because of the mechanism of metamodel pruning and aspect weaving based adaptation. Writing adaptations can be challenging depending on the developers' knowledge of the target input metamodels. Our approach is relevant if the number of transformations to be reused is significant. This means that the effort to convert the transformations is greater than writing adaptations of a metamodel. However, once the adaptation is done, the developers can reuse all model refactorings written for the original input metamodel. Conversely, if a developer specifies a new refactoring on the input metamodel, it can readily be applied on all target metamodels if adaptations are provided.

Although we show reuse of a kind of model transformation, namely, refactoring, we predict its extensibility to arbitrary model transformations with arbitrary input metamodels. In addition, our approach also fits well in the context of metamodel evolution. Indeed, all model transformations written for an old version of a given metamodel (for example, UML 1.2) can be reused for a new version (for example, UML 2.0) once the adaptation is done. Moreover, the models do not need to be migrated from an old version to a new one.

6 Related work

Reuse in MDE has not been sufficiently investigated as compared to object-oriented (OO) programming. However, we observe some efforts in the MDE community that are directly inherited from type-safe code reuse in OO programming and, in particular, from generic programming.

Generic programming is about making programs adaptable using generic operations that are functional across several input domains [22]. This style of programming allows writing programs that differ in their parameters, which may be either other programs, types and type constructors, class hierarchies, or even programming paradigms [22]. Aspects [23] and open-classes [18] are powerful generic programming techniques for adapting programs by augmenting their behavior in existing classes [24,25]. Other languages that provide support for generic programming are Haskell

and Scala [26]. The use of Haskell has been investigated [27] to specify refactorings based on high-level graph algorithms that could be generic across a variety of languages (XML, Pascal, Java), but its applicability does not seem to go beyond a proof of concept. Scala's implicit conversions [28] simulate the open-class mechanism in order to extend the behavior of existing libraries without actually changing them. Although Scala is not a *model-oriented* language, developers can build type-safe reusable model transformations on top of EMF because of its seamless integration with Java. However, it would require writing a significant amount of code and manage relationships among generic types.

In the MDE community, Blanc et al. propose an architecture called *Model Bus* that allows the interoperability of a wide range of modeling services [29]. The term '*modeling service*' defines an operation having models as inputs and outputs such as model editing, model transformation, and code generation. Their architecture is based on a metamodel that ensures type compatibility checking by describing services as software components having precise input and output definitions. However, the type compatibility defined in this metamodel relies on a simple notion of model types as sets of metaclasses, but without any notion of model type substitutability. Other works [30,31] study the problem of generic model transformations using a mechanism of parameterization. However, these transformations do not apply to different metamodels but to a set of related models.

Modularity in graph transformation systems was also explored [32]. In this area, an interesting work was done by Engels et al. who presented a framework for classifying and defining relations between typed graph transformation systems [33]. This framework integrates a novel notion of substitution morphism that allows to define the semantic relation between the required and provided interfaces of modules in a flexible way.

7 Discussion

In this paper, we combine ideas from two recently published papers on metamodel pruning [6] and manual specification of generic model refactoring [8]. In [8], the authors present an approach to manually specify generic model transformations and in particular refactorings. A generic metamodel is manually specified and a generic transformation is written for the generic input metamodel. Other target input metamodels are then adapted to the generic metamodel to achieve genericity and reuse. This approach is not applicable to legacy model transformations where we do not use a generic metamodel but an existing and possible large input metamodel such as UML. Adapting a target input metamodel to this large metamodel to make it a subtype is a very tedious task. It sometimes requires several unnecessary adaptations because many of the concepts may not be used in the transformation. We deal with this problem via metamodel pruning [6] in our work to automatically obtain the effective input metamodel which plays the role of the generic metamodel. This automatic synthesis of the effective input metamodel extends the approach in [8] to legacy model transformations written for arbitrary input metamodels. It also helps drastically reduce the number of required adaptations via aspect weaving and the time for type matching. Adaptation followed by verification using model typing, used in our approach, may be compared to generic pattern-matching techniques [34,35]. These pattern matching techniques can automatically detect concept similarities between metamodels. However, these similarities are often limited to the syntax of the metamodels and not their intended semantics. For instance, the notion of a Class may have very different meanings in two metamodels. Simply matching the concept Class and its structure in two metamodels does not ascertain a 100% conformance between these seemingly similar types. A number of ambiguities may crop up due to the same name but different structure of concepts while pattern matching. Human intervention is required to clearly build a bridge between concepts in two metamodels. The precise mechanism of adapting a metamodel using aspects followed by verification using model typing overcomes the limitations of classical pattern-matching mechanisms.

8 Conclusion

In this paper, we present an approach to make model transformations reusable across structurally different metamodels. This approach relies on metamodel pruning, model typing, and a mechanism of adaptation based mainly on the weaving of aspects. We illustrate our approach with the Pull Up Method refactoring and validate it for three different refactorings (Encapsulate Field, Move Method, and Pull Up Method) for three different industrial metamodels (Java, MOF, and UML) in a concrete application. We demonstrate that our approach ensures a flexible reuse of model transformations. We enlist the limitations of our approach based on the theoretical foundations of model typing [5]. We predict that our approach could be generalizable to arbitrary model transformations that can be used for various input domains such as the computation of metrics, detection of patterns, and inconsistencies. As future work, we plan to increase the repository of legacy transformations adapted to several different metamodels, in particular industry standards such as Java, MOF, and UML. We intend to apply our approach to the reuse of OCL constraints used as pre-/post-conditions of model transformations. We also plan to apply our approach on large industry-strength transformations. This will help us highlight the effectiveness and limitations of our approach. Finally, we are presently investigating a semi-automatic approach to adapt the effective input model to the target input model.

A Appendix

A.1 Kermeta Code for the Pull Up Method Refactoring

```
package refactor;

class Refactor<MT : EffectiveMM> {

  operation pullUpMethod(
                source : MT::Class,
                target : MT::Class,
                meth   : MT::Method) : Void

  // Preconditions
  pre sameSignatureInOtherSubclasses is do
        target.subClasses.forAll{ sub |
            sub.methods.exists{ op | haveSameSignature(meth, op) } }
  end

  // Operation body
  is do
      target.methods.add(meth)
      source.methods.remove(meth)
  end
}
```

Listing 1 Kermeta Code for the Pull Up Method Refactoring

A.2 Kermeta Code for Adapting the Java Metamodel

```
package java;

require "Java.ecore"

aspect class Classifier {
  reference inv_extends : Classifier[0..*]#extends
  reference extends : Classifier[0..1]#inv_extends
}

aspect class Class {

  property superClasses : Class[0..1]#subClasses
    getter is do
      result:=self.extends
    end

  property subClasses : Class[0..*]#superClasses
    getter is do
      result := OrderedSet<java::Class>.new
      self.inv_extends.each{subC|result.add(subC)}
    end
}
```

Listing 2 Kermeta Code for Adapting the Java Metamodel

A.3 Kermeta Code for Adapting the MOF Metamodel

```
package kermeta;

require kermeta

aspect class ParameterizedType {
 reference typeDefinition: GenericTypeDefinition[1..1]#
     inv_typeDefinition
}

aspect class GenericTypeDefinition {
 reference inv_typeDefinition: ParameterizedType[1..1]#
     typeDefinition
}

aspect class Type {
 reference inv_superType: ClassDefinition[0..*]#superType
}

aspect class ClassDefinition {

 reference superType : Type[0..*]#inv_superType

 property superClasses : ClassDefinition[0..*]#subClasses
  getter is do
    result := OrderedSet<ClassDefinition>.new
    self.superType.each{ c |
      var clazz : Class init Class.new
      clazz ?= c
      var clazzDef : ClassDefinition init ClassDefinition.new
      clazzDef ?= clazz.typeDefinition
      result.add(clazzDef) }
  end

 property subClasses : ClassDefinition[0..*]#superClasses
  getter is do
    result := OrderedSet<ClassDefinition>.new
    var clazz : Class
    clazz ?= self.inv_typeDefinition
    clazz.inv_superType.each{ superC | result.add(superC) }
  end
}
```

Listing 3 Kermeta Code for Adapting the MOF Metamodel

A.4 Kermeta Code for Adapting the UML Metamodel

```
package uml;

require "http://www.eclipse.org/uml2/2.1.2/UML"

aspect class Classifier {
 reference inv_general : Generalization[0..*]#general
}

aspect class Class {

 property superClasses : Class[0..*]#subClasses
  getter is do
    result := OrderedSet<uml::Class>.new
    self.generalization.each{ g | result.add(g.general) }
  end

 property subClasses : Class[0..*]#superClasses
  getter is do
    result := OrderedSet<uml::Class>.new
    self.inv_general.each{ g | result.add(g.specific) }
  end
}
```

Listing 4 Kermeta Code for Adapting the UML Metamodel

A.5 Kermeta Code for Applying the Pull Up Method Refactoring on the UML metamodel

```
package refactor;

require "http://www.eclipse.org/uml2/2.1.2/UML"

class Main {

 operation main() : Void is do

  var rep : EMFRepository init EMFRepository.new

  var model : uml::Model
  model ?= rep.getResource("lan_appli.uml").one

  var source: uml::Class init getClass("PrintServer")
  var target: uml::Class init getClass("Node")
  var meth  : uml::Operation init getOperation("bill")

  var refactor : refactor::Refactor<uml::UmlMM>
          init refactor::Refactor<uml::UmlMM>.new

  refactor.pullUpMethod(source, target, meth)
 end
}
```

Listing 5 Kermeta Code for Applying the Pull Up Method Refactoring on the UML metamodel

References

1. Biggerstaff, T.J., Perlis, A.J.: Software Reusability Volume I: Concepts and Models, vol. 1.. ACM Press, Addison-Wesley, Reading (1989)
2. Mili, H., Mili, F., Mili, A.: Reusing software: Issues and research directions. IEEE Trans. Softw. Eng. **21**(6), 528–562 (1995)
3. Basili, V.R., Briand, L.C., Melo, W.L.: How reuse influences productivity in object-oriented systems. Commun. ACM **39**(10), 104–116 (1996)
4. Blanc, X., Ramalho, F., Robin, J.: Metamodel reuse with MOF. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05), pp. 661–675 (2005)
5. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (2007)
6. Sen, S., Moha, N., Baudry, B., Jezequel, J.M.: Meta-model pruning. In: ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09). Springer, Berlin (2009)
7. Steel, J., Jézéquel, J.M.: On model typing. J. Softw. Syst. Model. (SoSyM) **6**(4), 401–414 (2007)
8. Moha, N., Mahé, V., Barais, O., Jézéquel, J.M.: Generic Model Refactorings. In: ACM/IEEE 12th International Conference on Model

Driven Engineering Languages and Systems (MODELS'09), Springer, Berlin (2009)
9. Fowler, M.: Refactoring – Improving the Design of Existing Code. 1^{st} edn. Addison-Wesley, Boston (1999)
10. Moha, N., Sen, S., Faucher, C., Barais, O., Jézéquel, J.M.: Evaluation of Kermeta on Graph Transformation Problems. Int. J. Softw. Tools Technol. Transf. (STTT) (2010) (To appear)
11. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theoret. Comput. Sci. **152**, 125–142 (2006)
12. Janssens, D., Demeyer, S., Mens, T.: Case study: Simulation of a LAN. Electron. Notes Theoret. Comput. Sci. **72**(4) (2003)
13. Hoffman, B., Pérez, J., Mens, T.: A case study for program refactoring. In: 4th International Workshop on Graph-Based Tools (GraBaTs'08) (2008)
14. OMG: MOF 2.0 core specification. Technical Report formal/06-01-01, OMG (2006)
15. OMG: The UML 2.1.2 infrastructure specification. Technical Report formal/2007-11-04, OMG (2007)
16. OMG: Architecture-driven modernization (ADM): Abstract syntax tree metamodel (ASTM) 1.0 - beta 2. Technical Report ptc/09-07-06, OMG (2009)
17. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MODELS'05), pp. 264–278. Springer, Berlin (2005)
18. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.D.: Multijava: Modular open classes and symmetric multiple dispatch for java. In: 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00), pp. 130–145 (2000)
19. OMG: The Object Constraint Language Specification 2.0. Technical Report ad/03-01-07, OMG (2003)
20. Bruce, K.B., Vanderwaart, J.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Electron. Notes Theoret. Comput. Sci. **20**, 50–75 (1999)
21. Kermeta: http://www.kermeta.org/. Accessed on April 2010
22. Gibbons, J., Jeuring, J., (eds.): Generic Programming. Proceedings of the IFIP TC2/WG2.1 Working Conference on Generic Programming, Kluwer Academic Publishers, Boston (2003)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: 11th European Conference on Object-Oriented Programming (ECOOP'97), vol. 1241, pp. 220–242. Springer, Berlin (1997)
24. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. SIGPLAN Not. **37**(11), 161–173 (2002)
25. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: 27th International Conference on Software Engineering (ICSE'05), pp. 49–58. ACM Press, New York (2005)
26. Oliveira, B., Gibbons, J.: Scala for generic programmers. In: Hinze, R., Syme, D., eds.: SIGPLAN Workshop on Generic Programming (WGP'08), pp. 25–36. ACM Press, New York (2008)
27. Lämmel, R.: Towards generic refactoring. In: 3rd SIGPLAN Workshop on Rule-Based Programming (RULE'02), pp. 15–28. ACM Press, New York (2002)
28. Odersky, M., et al.: An overview of the Scala programming language. Technical Report IC/2004/64. EPFL Lausanne, Switzerland (2004)
29. Blanc, X., Gervais, M.P., Sriplakich, P.: Model Bus : Towards the interoperability of modelling tools. In: European Workshop on Model Driven Architecture: Foundations and Applications (MDAFA'04). Volume 3599 of LNCS, pp. 17–32. Springer, Berlin (2004)
30. Amelunxen, C., Legros, E., Schurr, A.: Generic and reflective graph transformations for the checking and enforcement of modeling guidelines. In: IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC'08), pp. 211–218, IEEE Computer Society, Washington, DC (2008)
31. Münch, M.: Generic Modelling with Graph Rewriting Systems. PhD thesis, Berichte aus der Informatik. RWTH Aachen, Aachen (2003)
32. Heckel, R., Engels, G., Ehrig, H., Taentzer, G.: Classification and comparison of module concepts for graph transformation systems. In: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools, pp. 639–689. World Scientific Publishing Co., Inc., Hackensack (1999)
33. Engels, G., Heckel, R., Cherchago, A.: Flexible interconnection of graph transformation modules. In: Formal Methods in Software and Systems Modeling. vol. 3393 of LNCS., pp. 38–63. Springer, Berlin (2005)
34. Lahire, P., Morin, B., Vanwormhoudt, G., Gaignard, A., Barais, O., Jézéquel, J.M.: Introducing variability into aspect-oriented modeling approaches. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07). LNCS, pp. 498–513. Springer, Berlin (2007)
35. Ramos, R., Barais, O., Jézéquel, J.M.: Matching model-snippets. In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS'07). LNCS, pp. 121–135. Springer, Berlin (2007)

Author Biographies

Sagar Sen is a PhD (2010) in computer science from University of Rennes 1, France. He obtained his M.Sc. in computer science from McGill University, Montreal. Currently, he is postdoctoral researcher at INRIA team PULSAR at Sophia-Antipolis, France. His interests are in model-driven engineering, models@runtime, software product lines and scaling of formal methods to tackle validation in large software systems. Currently, he is working on applying models@runtime and dynamic adaptation in the area of computer vision systems. He has published 6 journal papers and 16 peer-reviewed international conference papers.

Naouel Moha received the Master degree in computer science from the University of Joseph Fourier, Grenoble, in 2002. She also received the PhD degree, in 2008, from the University of Montreal (under Professor Yann-Gael Gueheneuc's supervision) and the University of Lille (under the supervision of Professor Laurence Duchien and Anne-Francoise Le Meur). The primary focus of her PhD thesis was to define an approach that allows the automatic detection and correction of design smells, which are poor design choices, in object-oriented programs. She has been a postdoctoral researcher in the INRIA team-project Triskell. Currently she is an associate professor with University of Rennes 1 and an adjoint assistant professor at University of Quebec in Montreal. Her research interests include software quality and evolution, in particular refactoring and the identification of patterns in service oriented systems.

Vincent Mahé received the Master degree in industrial economics from the University of Rennes 1 in 1991. After a first career in a software company, he received a Master degree in computer science form the University of Rennes 1 in 2006. He now works as research engineer in the AtlanMod team. His research interests covers Model-Driven Engineering with focus on generic tooling and design of domain specific metamodels.

Olivier Barais Associate Professor, Université de Rennes 1/ IRISA, born in 1980 Olivier Barais received an engineering degree from the Ecole des Mines de Douai, France in 2002 and a PhD in computer science from the University of Lille 1, France in 2005. After having been a PhD student in the Jacquard INRIA research team, he is currently associate professor at University of Rennes 1 and a member of the Triskell INRIA group. His research interests include Component Based Software Design, Model-Driven Engineering and Aspect Oriented Modelling. Olivier Barais has co-authored 6 journals, 30 international conference papers, 2 book chapters and 23 workshop papers in conferences and journals such as SoSym, IEEE Computer, ICSE, MoDELS and CBSE.

Benoit Baudry received his PhD in computer science from the University of Rennes, France in 2003. He first worked at CEA (French government nuclear agency) before joining INRIA in 2004. He is now a researcher in software engineering in the Triskell team at INRIA Rennes Bretagne Atlantique. In 2008 he was an invited scientist at Colorado State University. His research interests include software testing, aspect-oriented software development, model transformation and model-driven development. He is the vice-chair of the steering committee of the International Conference on Software Testing Verification and Validation. He is a member of the IEEE and the IEEE Computer Society.

Jean-Marc Jézéquel is a Professor at the University of Rennes and the leader of an INRIA research team called Triskell. His interests include model driven software engineering for software product lines, and specifically component based, dynamically adaptable systems with quality of service constraints, including reliability, performance, timeliness etc. He is the author of several books published by Addison-Wesley and of more than 100 publications in international journals and conferences. He is a member of the steering committees of the AOSD and MODELS conference series. He also served on the editorial boards of IEEE Transactions on Software Engineering and on the Journal on Software and System Modeling and the Journal of Object Technology. He received an engineering degree in Telecommunications from ENSTB in 1986, and a PhD degree in Computer Science from the University of Rennes, France, in 1989.

Integrating Legacy Systems with MDE*

Mickael Clavreul
INRIA
Campus de Beaulieu
35042 Rennes Cedex, France
mickael.clavreul@inria.fr

Olivier Barais
IRISA, Université Rennes 1
Campus de Beaulieu
35042 Rennes Cedex, France
barais@irisa.fr

Jean-Marc Jézéquel
IRISA, Université Rennes 1
Campus de Beaulieu
35042 Rennes Cedex, France
jezequel@irisa.fr

ABSTRACT

Integrating several legacy software systems together is commonly performed with multiple applications of the Adapter Design Pattern in OO languages such as Java. The integration is based on specifying bi-directional translations between pairs of APIs from different systems. Yet, manual development of wrappers to implement these translations is tedious, expensive and error-prone. In this paper, we explore how models, aspects and generative techniques can be used in conjunction to alleviate the implementation of multiple wrappers. Briefly the steps are, (1) the automatic reverse engineering of relevant concepts in APIs to high-level models; (2) the manual definition of mapping relationships between concepts in different models of APIs using an ad-hoc DSL; (3) the automatic generation of wrappers from these mapping specifications using AOP. This approach is weighted against manual development of wrappers using an industrial case study. Criteria are the relative code length and the increase of automation.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design, Reliability

Keywords

Legacy Systems, Aspects, Models, MDE

1. INTRODUCTION

As development techniques, paradigms, technologies and methods are evolving far more quickly than domain applications, software evolution and maintenance is a constant challenge for software engineers.

In a context of Enterprise Application Integration (EAI), a key concern is the translation of the outputs of some applications into inputs for other applications. Text-based formats can readily be handled with parsers, interpreters and text-based adapters. Even more easily, XML-based formats are nowadays supported by a wide range of tools or methods that help people convert information for a particular use, platform or software. The problem is quite different when one has to efficiently translate a continuous flow of data from one legacy API (Application Programming Interface) to another one. A possible solution could be based on an application of the Adapter Pattern [20] to map one call in API I_1 to slightly different calls that cope with API I_2, hence "*wrapping*" I_2 in such a way that it offers an interface compatible with I_1.

Multiple occurrences of the Adapter Pattern are then needed to integrate applications in such a way that they mutually inter-operate, including with previous versions of themselves. Even if a kind of *intermediate* API could be used to reduce the number of adapters from $2 * n * (n-1)$ down to $2 * (n+1)$ for n applications, that still leaves us with a lot of tedious and error-prone adaptation code to be developed when n is large. In some domains, such as the management of heterogeneous on-line equipments for digital video broadcasting, a steady flow of both requirements and third party new products makes it very hard to both keep up with evolutions and still guarantee backward compatibility and interoperability between versions: even the intermediate language has to be constantly refined, leading to expensive maintenance operations.

The contribution of this paper is to propose a model-driven approach to alleviate the implementation of multiple wrappers in that kind of context. Our approach is technically based on three steps: (1) the automatic reverse engineering of relevant concepts from APIs to high-level models; (2) the manual definition of mapping relationships between concepts in different models of APIs using an *ad hoc* Domain Specific Language (DSL); (3) the automatic generation of wrappers from these mapping specifications using Aspect Oriented Programming (AOP) to avoid changes in legacy APIs. Our proposition highlights how models, aspects and generative techniques might be used in conjunction with legacy code to reduce costs and increase efficiency in software development. The remainder of this paper is organized as follows. Section 2 introduces a motivating example to illustrate our approach. Section 3 outlines the global process of our solution. Section 4 illustrates the first step of our approach and describes how we use reverse-engineering to create API-specific models. Section 5 and Section 6 refer to the second step of the process and describe the mapping lan-

*This work has been partially supported by the MOPCOM-I Project from the Competitiveness Cluster of Brittany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

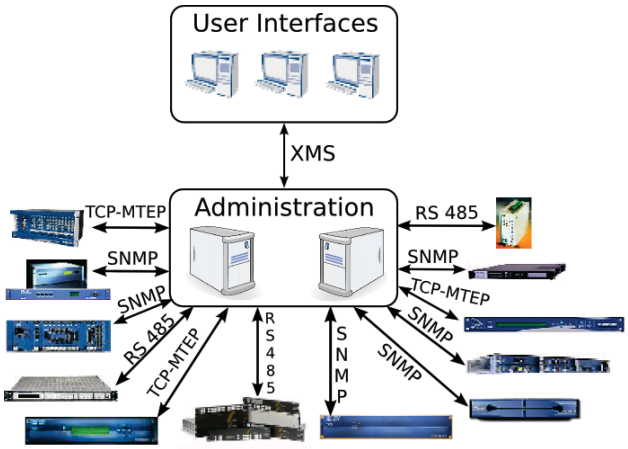


Figure 1: Global View of the Management Architecture with some examples of managed physical devices

guage, how it is used and how mappings implementation is achieved. Section 7 deals with the automatic generation of wrappers from mappings specifications using aspect-oriented techniques. Section 8 compares our solution on the case study to the manual implementation of adapters, in terms of effort. Section 9 discusses articles and ideas related to our work. Finally, Section 10 concludes this paper.

2. MOTIVATING EXAMPLE

2.1 General Description

As a motivating example, we are going to consider the domain of configuring and managing heterogeneous equipments for video and broadcasting, such as Thomson Extensible Management System for Digital TV. This management system deals with the intercommunication of legacy hardware devices (*i.e.* Network Adapters, MPEG Multiplexers, Encoders, Decoders,...). As shown in Figure 1, digital devices are designed by different manufacturers and from different technologies. Each one provides a specific API for management purposes. For management and configuration concerns, Thomson provides a distributed architecture with a set of remote user interfaces and administration servers that communicate with each other through an intermediate API called XMS. Administration servers main responsibility is thus to convert XMS orders into device-specific ones as shown in Figure 1.

To allow integration with existing platforms and systems, Thomson has been developing APIs for an extensive set of protocols such as MTEP, SNMP, XMS, TCP/IP, RS232/485 and so on (see Figure 1). The evolution of legacy equipments induces the development and the maintenance of several versions of APIs, both for the specific protocols and for the intermediate language XMS. This situation leads to the combinatorial explosion of the number of wrappers to be developed.

2.2 Translation Issues

In Figure 2 we present the specification of the mappings between objects from the MTEP API to the XMS API. The MTEP specification defines groups of devices. Devices are either nominal or redundant. Nominal devices are preferably used by the system to perform some treatment. If nominal

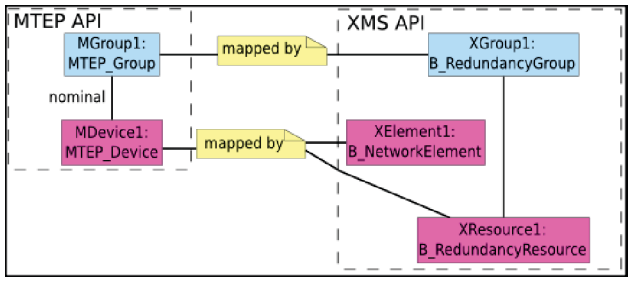


Figure 2: Example of Objects Diagrams with Mapping Information

devices are not available because of bugs or failures, the system uses redundant devices that offer equivalent functionalities. While the MTEP API qualifies the redundant or nominal characteristic in the form of a relation between a group and a device, the XMS API proposes two different objects. XMS defines nominal devices as B_NetworkElement and redundant devices as B_RedundancyResource. The concept of group (B_RedundancyGroup) is available only for redundant devices.

The semantics of "*mapped by*" is informally defined at the level of the specification while experts from the domain really define the semantics of "*mapped by*" at the code level only (see Listing 1). This situation often results in ambiguous interpretations of the mappings. Therefore we have a need to specify the formal semantics for the "*mapped by*" relation. The personal interpretation by the developers is critical to create wrappers for the API objects. The translation of MGroup1 to XGroup1 (see Figure 2) consists in copying all data from MGroup1 attributes to XGroup1 equivalent attributes. The translation of MDevice1 is more complicated since the semantics of *mapped by* does not provide enough information for the data transfer. The developers may create XElement1 or XResource1 or both. Moreover developers are not able to infer the sequence of the translation process from the mappings specification. For instance, developers could either implement the translation of MGroup1 first or start with MDevice1.

To summarize, we observe three issues in the current specification of the translation:

- The mapping descriptions are often ambiguous.
- Developers often introduce implementation bugs while coding wrappers for such specification.
- The automatic synthesis of the translation process from the specification is not possible.

3. GLOBAL SOLUTION PROCESS

We propose a semi-automated process to limit the ambiguity of mapping descriptions, to reduce bugs in the implementation and to automate the design of the translation process. The process is composed of four steps as illustrated on Figure 3:

① **- Model Abstraction from Legacy Code:**
We analyze the legacy code of the APIs to find all relevant classes. We automatically build an application model with the use of reverse engineering techniques (see Section 4).

```
class: Mtep_Device
---------- First object to create ----------
mapped class: B_RedundantResource
read attributs:
 comment (String)⇒B_RedundantResource.name(String)
read associations:
  EMPTY
---------- Second object to create ----------
mapped class: B_NetworkElement
read attributs:
 device_id (int)  ⇒ B_NetworkElement.deviceId(int)
 type (short)     ⇒ B_NetworkElement.type(short)
 extended_type
  (short)     ⇒ B_NetworkElement.extendedType(int)
 comment (String) ⇒ B_NetworkElement.name(String)
read associations:
      ...
```

Listing 1: Example of mapping instructions provided by experts between a MTEP_Device and XMS corresponding elements: B_NetworkElement and B_RedundantResource

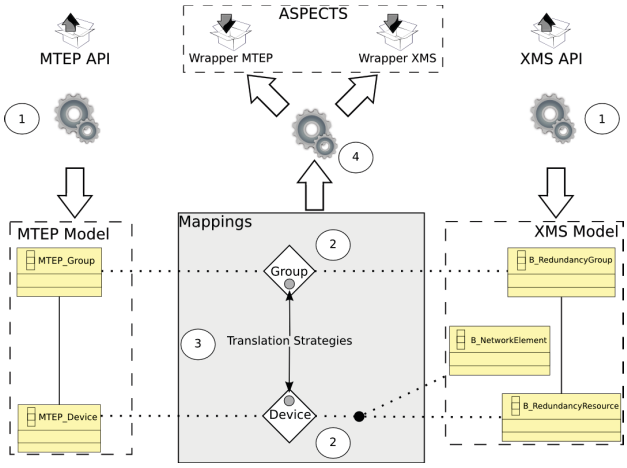


Figure 3: Our process is composed of four steps. We automatically extract models from the APIs. Users define mappings and select strategies for translating model elements. We automatically generate adapters for each API.

② - High Level Mapping Description:
Users define mappings at the model level between classes from the MTEP API and classes from the XMS API and vice versa (see Section 5). In Figure 3, white diamonds represent the mapping relations between classes from the APIs. Big black dots pinpoint that a mapping relation has multiple inputs or outputs. See Section 5 for more details on the mapping language.

③ - Translation Strategies:
Users choose strategies of translation to specify how the data should be transfered between model elements (see Section 6).

④ - Generation of Bidirectional Adapters:
We automatically generate bidirectional wrappers as aspects. We filter the information from the models and the mappings to generate code. We generate code only for the wrappers to avoid invasion of the legacy code.

4. MODEL ABSTRACTION FROM LEGACY CODE

In a context of Enterprise Application Integration (EAI), a key concern is the translation of the outputs of some applications into inputs for other applications. The problem is to efficiently translate a continuous flow of data from one API (Application Programming Interface) to another one. A possible solution could be based on an application of the Adapter Pattern [20] to map one call in API I_1 to slightly different calls that cope with API I_2, hence "*wrapping*" I_2 in such a way that it offers an interface compatible with I_1. We propose to move the mapping specification from the code level to the model level. The basic idea is to automatically build a model from the code of the API. The first step is to detect the relevant model elements that are valuable in terms of domain representativeness. We identify these model elements by analyzing the structural data (i.e OO classes) of APIs.

From this structural information, we build a model of the application. The model of the application conforms to a high-level representation of the implementation language, i.e a meta-model. We filter the model to remove any unnecessary information.

The model we build only contains domain related model elements we want to align with the model elements of another API.

We performed the reverse engineering of the API code (see Listing 2) with SpoonEMF[1]. SpoonEMF offers a Java code analyzer that automatically produces a model of the code as an Abstract Syntax Tree (AST) (see Figure 4).
We filter the AST to remove irrelevant data and build a new model that conforms to the ECore[2] formalism. We choose the ECore formalism because it is the input of the tools we use in further steps of our process.

For instance, we analyze the Java code (see Listing 2) of the MTEP API to create the corresponding application model (see Figure 4). Through the analysis, we found *device_id* and *comment* are attributes of a Java class called "Mtep_Device" and $inputFromBuffer$ is a method of the same class. Our interest focus on structural data. Therefore we keep the attributes and drop the method *inputFromBuffer*. In a second time, we look for getter and setter methods to identify read-only or read-write attributes. Third step is to create the corresponding ECore class as shown in Figure 5.

5. HIGH LEVEL MAPPING DESCRIPTION

We propose to move the mapping descriptions from an informal text-based representation to a formal specification. We adapted and extended the formalism introduced in [11] to provide users with a graphical language. Hausmann *et al*'s formalism includes elements to express *consistency between models* or in another way *the conditions under which two models are compatible*.

We kept the following definitions (see Figure 6) to express various configurations of mapping:

- A white diamond indicates a mapping definition and has a name.

[1]http://spoon.gforge.inria.fr
[2]http://www.eclipse.org/modeling/emf/?project=emf

```
package mtep.mtep_9_20.entity.dmt.impl;
import ...
public class Mtep_Device extends MtepElement
             implements MtepElementInterface
{
    private int device_id;
    private String comment;

    public Mtep_Device(){}

    public int getDevice_id(){return device_id;}

    public void setDevice_id(int aDevice_id)
    {device_id = aDevice_id;}

    public String getComment() {return comment;}

    public boolean inputFromBuffer(InputStream buffer)
             throws ExConversionProblem,ExInvalidFormat
    {return true ;}
    ...
}
```

Listing 2: Extract of MTEP Device Java class

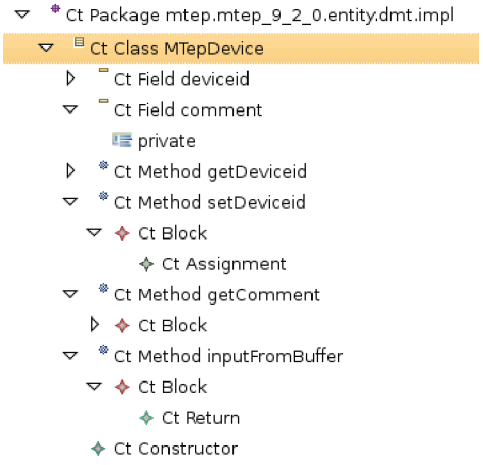


Figure 4: Application Model we got from the reverse engineering of the MTEP_Device Java class

- A model element is linked with a mapping definition with a dotted line.
- If a mapping definition involves multiple sources or multiple targets, a black circle is put between the mapping description and the set of model elements involved.

We propose to extend the current formalism (see Figure 7) to include the following concerns explicitly:

- We explicitly separated mapping descriptions into four types. Each type depends on the multiplicity of sources and targets model elements that we want to map.

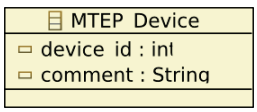


Figure 5: MTEP_Device Class in the ECore model produced from code

- We allow the creation of a mapping description between any model element such as classes (i.e real objects), attributes, and relations between model elements.
- We propose that each mapping description is bound to a strategy. A strategy is converted into a specific translation algorithm. We observed that a lot of translations share the same behavior and may be reused for several model elements translation. We conclude that users could use predefined strategies for common translation and only provide their own when necessary:
  - CloneStrategy is fully-automatic and consists in copying source attributes to target attributes with no changes
  - RenameStrategy is semi-automatic and takes data from users to align concepts by their names
  - ConstrainedStrategy is semi-automatic and provides a limited model-alignment language. The language offers atomic operations on models that guarantee the termination and the bijectiveness of the translation.
  - FreeStrategy is manual and relies on a Turing-complete language for arbitrary complex mappings.

Figure 6 illustrates the mapping language and the use of strategies on the MTEP to XMS translation example. This example is based on the mapping descriptions provided by experts to translate a device from MTEP to XMS. We create graphical mapping descriptions and link them to model elements from APIs. In the example we map a MTEP_Device to both B_NetworkElement and B_RedundancyResource and we associate a strategy of translation. The strategy of translation is contained in the mapping and is represented by a small light-grayed circle. The strategy we use in the translation is a RenameStrategy because we want to align the MTEP API model elements and the XMS API model elements on names. A RenameStrategy needs some additional input from users to automatically create a wrapper that works. We put such input in the light-grayed rectangle that is linked to the strategy. Inputs are defined with the directive language used in [9].

6. TRANSLATION STRATEGIES AND OPERATIONAL SEMANTICS DEFINITION WITH KERMETA

From this point, we manipulate one model for each API and an additional mapping model independently. We have to create wrappers to convert the API I_1 to the API I_2. It is not yet possible to automatically generate these wrappers because we need to know how to interpret the mapping language and the relation between the mapping language and the APIs model elements. We have to combine the model elements from the APIs and the mapping language elements to create a new language that describes how to execute the mappings. We give the operational semantics of this new language with Kermeta, which thus plays the role of the meta-language in which our mapping language is defined. The FreeStrategy presented in Section 5 can then be considered as an escape mechanism to the meta-language.

Kermeta [14] provides a way to compose meta-models concepts declarations using Aspects at the modeling level. This

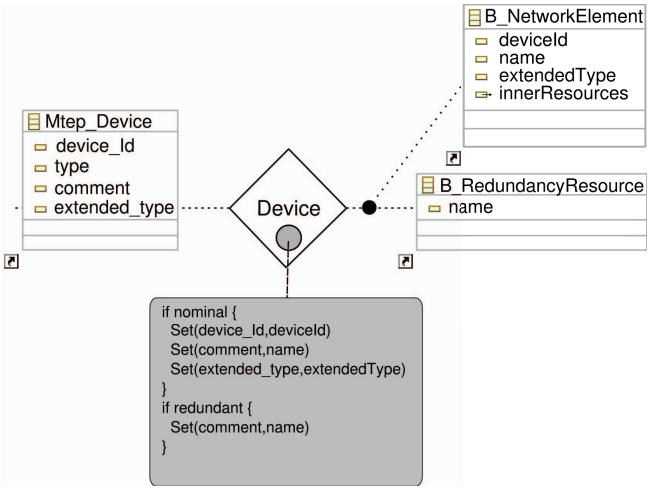


Figure 6: Graphical description of mappings between mtep and xms elements.

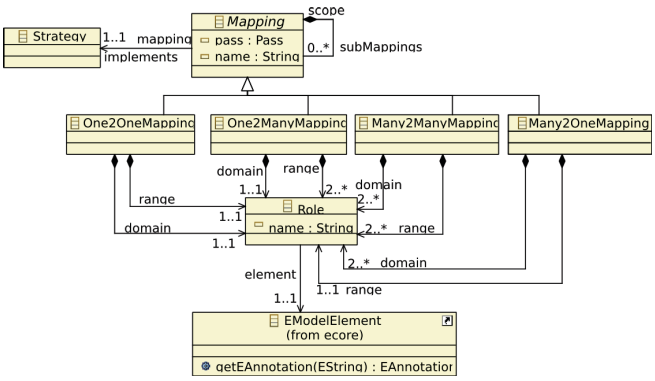


Figure 7: Graphical Mapping Language Specification as a meta-model

composition allows developers to manipulate concepts from different meta-models. The Kermeta language has also been specifically developed to express DSL operational semantics. We are then able to describe how the mapping language and its syntactic elements have to be implemented. For instance, we describe the semantics of mappings, the semantics of the relations with API model elements, and the semantics of the strategies.

To illustrate the use of Kermeta, we present the implementation of two strategies with the Thomson's case study as a background. Listing 3 show the implementation of the CloneStrategy and Listing 4 show one implementation of the FreeStrategy.

6.1 CloneStrategy Automation for Simple Mappings

The default translation strategy of our process is the CloneStrategy. This strategy needs to be fully automated to alleviate users efforts in designing the translation between APIs. We propose a Kermeta Visitor approach to create main concepts and their attributes. Listing 3 is an example of the implementation of the strategy. The implementation is achieved through two methods: "toDomain()" method literally "clone" a class from the first API to create a class from the second API; "toRange" method is the reverse.

```
aspect class CloneStrategy {
 operation toDomain() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  m.range.element.getMetaClass().ownedAttribute
    .each{ra|
     m.domain.element.getMetaClass().ownedAttribute
       .each{da|
        if da.name.equals(ra.name) then
         m.domain.element.getMetaClass().~set(
           da,m.range.element.getMetaClass().get(ra)
         )
        end
     }}
 end
 operation toRange() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  m.domain.element.getMetaClass().ownedAttribute
    .each{da|
     m.range.element.getMetaClass().ownedAttribute
       .each{ra|
        if ra.name.equals(da.name) then
         m.range.element.getMetaClass().~set(
           ra,m.range.element.getMetaClass().get(da)
         )
        end
     }}
 end
}
```

Listing 3: Example of the CloneStrategy in Kermeta language

6.2 Bidirectional FreeStrategy for Complex Mappings

Some mappings cannot be achieved by using available strategies such as CloneStrategy, RenameStrategy, or ConstrainedStrategy. Users need some complex algorithms to create corresponding model elements so they use the full expressiveness of the Kermeta language to define the translation. For simplicity's sake, we present only one type of XMS network device that is a B_NetworkElement. There exists four other types of network devices the translation has to consider. The type of a device is given by an integer in the MTEP API whereas each type of device is a different class in the XMS API. Listing 4 shows the implementation of the strategy to perform a search-and-test activity that creates various types of devices. To conclude this subsection, FreeStrategy are very useful to handle very specific translations. Moreover we are able to produce new custom strategies that can be reused in further developments.

7. BIDIRECTIONAL ADAPTERS GENERATION

The generation of the wrappers is the last step of the global process. This step is similar to well-known generative techniques that produce code from models. The code generator uses two input parameters: the mapping design model and the Kermeta code that supports the translation strategies. We adapted code generation methods to use aspects-weaving techniques, i.e, we generate wrappers as aspects to avoid the invasion of the original code of APIs. The generation step is composed of three stages (see Figure 8):

- We use the Kermeta Compiler to merge the structural definitions from the APIs with the behavioral definitions from the mappings and strategies. The output is an ECore model.

```
class FreeStrategyDevice inherits FreeStrategy {
 reference dom : Mtep_Device
 reference ran : B_NetworkElement

 operation toDomain() : Void is do
  var m : Mapping
  m ?= self.mapping
  self.dom := m.domain.element
  self.ran := m.domain.element

  if self.ran.isKindOf(B_Switcher) then
   self.dom.type := 0
  else if self.ran.isKindOf(B_TsProbe) then
    var ne : B_TsProbe
    ne ?= self.ran
    ne.tsProbeInputName := self.dom.device_Id
                           .substring(ne.tsProbeInputName.size-1,1)
                           .toInteger
    self.dom.type := 1
        ...
  end end

 operation toRange() : Void is do
  var m : One2OneMapping
  m ?= self.mapping
  self.dom := m.domain.element
  self.ran := m.domain.element

  if self.dom.type == 0 then
   self.ran := B_Switcher.new
  else if self.dom.type == 1 then
    var ne : B_TsProbe init B_TsProbe.new
    ne.tsProbeInputName := "TSProbe_" +
       self.dom.device_Id.toString
    self.ran := ne
   ...
 end end
}
```

Listing 4: Example of FreeStrategy for translating a MTEP_Device to the right type of B_NetworkElement and vice versa

- We filter the ECore model to remove all data that are in conflict with existing APIs. We obtain an ECore model that only contains the definitions related to the translation.
- We use an AspectJ generator written in Kermeta to generate aspects from the ECore model we filtered.

7.1 Merging Structure, Mappings and Strategies

We convert the mapping specifications into a Kermeta model. Each mapping is converted into two Kermeta aspects: one for the source model elements of the API I_1 and one for the target model elements of the API I_2. These aspects contain Kermeta operations that encapsulate the adaptation between the two APIs. We analyze strategies and additional alignment directives provided by users and translate them into Kermeta code. This code is the actual body of the methods previously created.

As an example, we consider the conversion of a MTEP_Device to a B_NetworkElement. We create two Kermeta aspects for both MTEP_Device and B_NetworkElement. We analyze the strategy associated to this mapping and produce corresponding Kermeta code (see Listing 4). We combine the definitions of the *device_id* and *comments* attributes (see Figure 5) with this additional behavior (*toDomain*() and *toRange*() operations) to produce the final adapter. Result is shown on Listing 5.

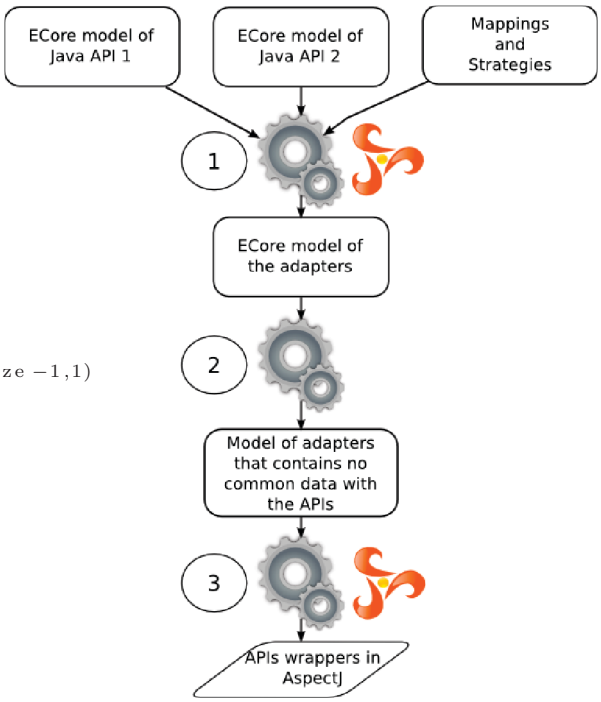


Figure 8: The production of executable code is composed of three steps. The Kermeta Compiler merges structural and behavioral definitions. We filter the resulting model. We generate the wrappers between APIs as aspects.

7.2 Avoid Invasion by Filtering

One key concern of our approach is to be non-invasive regarding legacy API code. The merged model we get from previous stage contains definitions of both API classes and wrappers. The generation of code from this model would build a set of new classes that would overwrite legacy classes. We avoid classes to be overwritten by removing class definitions that are equivalent between the models of the APIs and the model of the wrappers. We apply a filtering method that only keeps new class members or additional utility classes.

The method checks names of classes and their signature [9] to identify equivalent class definitions that we remove from the merged model. We get a new ECore model that only contains the methods and classes that define the wrappers between one API and another. In the example of the

```
aspect class MTEP_Device {
 /* Attributes from MTEP_Device class */
 attribute device_id : int
 attribute comment : String

 /* Kermeta code from Listing 4 */
 operation toDomain() : Void is do
  ...
 end
 operation toRange() : Void is do
  ...
 end
}
```

Listing 5: Aspect in Kermeta: combination of legacy data and adapters data

```
//classes exist in Legacy
classesinMM1 ?= generator.allClasses
//All classes
classesinCompiledEcore ?= generator1.allClasses
var classExistInLegacy : Sequence<EClass>
classExistInLegacy := classesinCompiledEcore.select{
 c| classesinMM1.exists{c1 | c1.name == c.name}
}
classExistInLegacy.each{c |
 var c1 : EClass init classesinMM1.detect{c2 |
  c2.name == c.name}
 /* Operations that should be introduced in a legacy
  * classes (operation that are in the class of the
  * compiled ecore but not in the original ecore */
 var operationToIntroduce : Sequence<EOperation> init
 //match by signature
 c.eOperations.select{op | not c1.eOperations.exists{
  op1 | op1.name == op.name}}
 /*Fields that should be introduced in a legacy
  *classes (fields that are in the class of the
  *compiled ecore but not in the original ecore*/
 var fieldsToIntroduce :
   Sequence<EStructuralFeature> init
    c.eStructuralFeatures.select{f |
     not c1.eStructuralFeatures.exists{f1 |
      f1.name == f.name}}
 //Generate aspects
 generateAspect(c,fieldsToIntroduce,
  operationToIntroduce, outputFolder)
}
 //newclasses => Standard POJO Generation
 classExistInLegacy.each{c |
  classesinCompiledEcore.remove(c)}
 classesinCompiledEcore.each{c|
  generatePojos(c,outputFolder)}
```

Listing 6: Kermeta code for filtering models elements that are in conflict with existing one in legacy APIs

MTEP_Device, the filtering process removes elements that exist in legacy APIs. As a consequence, *device_id* and *comment* attributes are removed from the definition of the MTEP_Device aspect.

7.3 Executable Code Production

We process the model of aspect definitions we got from the previous stage. The Kermeta compiler uses code templates to produce AspectJ executable code. Classes that do not exist in the legacy code are created whereas classes that already exists are augmented with inter-type declarations. The inter-type declarations encapsulate the translation behavior between existing classes. Adapters between the two APIs (see an example for a MTEP_Device in Listing 7) are composed with the original legacy code (available as a JAR) at load-time using the AspectJ compiler. Load-time weaving is deferred until the point that a class loader loads a class file and defines the class to the JVM. As a consequence, additional capabilities we brought through the adapters production does not pollute existing code embedded in legacy APIs.

8. DISCUSSION

As an evaluation of our solution, we propose to compare efforts between classic development techniques (followed by domain experts) and using our semi-automated process. As a benchmark, we are comparing our solution to Thomson Extensible Management System evolution on the specific example of MTEP to XMS protocol translations.

8.1 Impact of Automation on Wrappers Production

```
package net.thomson.protocol.mtep;
aspect Mtep_DeviceAspect {
 declare parents: Mtep_Device
   implements kermeta.language.structure.Object;

 public Mtep_Device_Input inputsLinkedWithBoards;
 public Mtep_Device_Output outputsLinkedWithBoards;
 public B_NetworkElement output;
 public B_RedundancyReplaceableResource outputRedundant;
 public void  mtep2xmsPass1(){
   ...
 }
 public void  mtep2xmsPass2(B_XmsBusinessData xbd,
    Mtep_Device.Object out, JavaBoolean nominal){
    ...
 }
}
```

Listing 7: AspectJ code produced by the Kermeta compiler for the MTEP_Device example

The case study is based on a subset of Thomson MTEP to XMS conversion. This subset contains seven MTEP-related concepts and twenty XMS-related concepts defined in their respective APIs.

From the correspondence specifications provided by experts, Thomson developers implemented twenty mappings to carry out the bijective translations between MTEP and XMS (see Figure 9 for details about mappings ratio). The application of our process on the same case study involves only eight bidirectional mappings, whose distribution is represented in Figure 10.

We can draw two conclusions from these figures:

1. We needed fewer mapping descriptions to handle the same case study.

2. Mapping descriptions complexity is reduced since Many-to-many mappings are not used and 75% of mapping descriptions are One-to-one mappings.

The first point comes from the use of a more adequate DSL to express mappings at the right level of abstraction. At the code level, developers implement complex mappings as one-to-one mappings because the implementation language do not offer higher-order mapping operators. The mapping language we propose offers more expressiveness to declare mappings so some of them, identified by experts, are not expressed anymore as single mappings but are encapsulated into higher-order mappings. The second point is related to the expressiveness of the DSL versus the implementation practices in Java. We observed that when people use low-level correspondence languages, they are more tempted to violate implementation practices of the Visitor Pattern to access incidental information (information from multiple concepts that do not take part in the original described mappings). Our approach limits this problem since the mapping DSL offers higher abstractions to retrieve data in a proper way: users are able to describe relations between mappings, so concepts involved in a mapping are confined to the original inputs and outputs of the wrappers.

Figure 11 is to be compared with Thomson implementation of adapters (100% of strategies would be FreeAlignment). This figure shows that most of strategies (63%) used to handle the case study are automatic or semi-automatic. Of course, it is not possible to automatically define all mapping implementations: that is why we provide a way to use

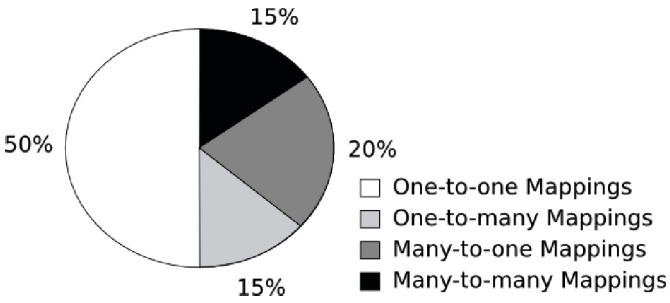


Figure 9: Distribution of the twenty mappings identified by the experts of the domain and implemented with Java

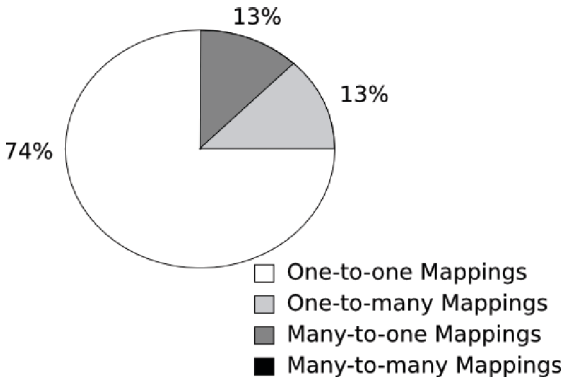


Figure 10: Distribution of the eight mappings identified by experts of the domain and implemented with the mapping language presented in Section 5)

a more powerful language to implement the remaining mappings.

These results are a first indication, on a relatively small example, that the use of a high-level language for mapping descriptions helps reduce the number of adapters to be implemented. It also gives additional evidence that the use of generative techniques cuts down global complexity and effort to produce adapters.

8.2 Effort Comparison

The second stage of our evaluation deals with effort estimation in terms of the number of lines of code (LOC). Thomson global adapter size for the case study is about 5350 LOCs to realize the bi-directional translation between MTEP and XMS protocols. Our approach is implemented using only 510 Kermeta LOCs. The effort has been evaluated to 136 hours of person work for the manual implementation of a

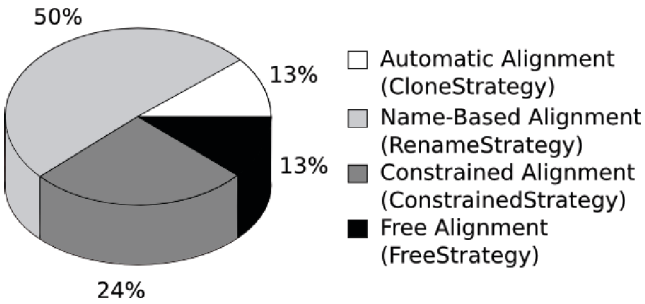


Figure 11: Ratio of strategy types used to implement adapters through the process we propose

Production of a new adapter	Manual Approach		Generative Approach	
	v1	v2 (avg)	v1	v2 (avg)
Code length (LOC)	5350	-	570	-
Total TDEV (Hours)	136	**+57**	150-200	**+9**

Table 1: Effort for manual and generative approaches for the production of a new adapter. The production of a second version (v2) increases the effort by an average of 57 hours for the manual approach and by an average of 9 hours for the generative approach.

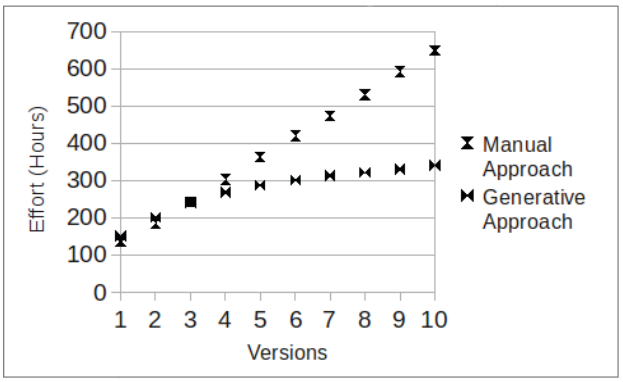


Figure 12: Cumulative effort for the production of new versions of adapters using manual or generative approach: effort (time of development in month) is on the y-axis and versions on the x-axis.

new adapter, compared to 150 hours to handle the same example with our approach. Considering up to 50 extra hours to take into account the introduction of a new mapping language and a new language for strategies definition for users, the effort needed to use our process is of 150 up to 200 hours (see Table 1) for the very first version of an adapter, which is slightly more than the manual approach.

However, the mean of the effort to produce a new version of an adapter for both approaches are 57 hours for a manual implementation versus 9 hours with the semi-automatic process.

These results have several consequences: First, we are able to say that our approach needs less manual implementation from users. Second, thanks to generative techniques, we were able to reduce the number of bugs in code and thus time spent in debugging has been drastically reduced. These improvements allow users to save some maintenance effort on the code in further evolutions. Figure 12 illustrates the effort reduction in the production of ten successive versions of this adapter. A manual approach induces a constant effort to develop and test new adapters versions. Our semi-automatic approach is expensive on the very first version (learning overhead and complex mapping definitions) but costs decrease with time as learning overhead decreases in further evolution. Even though benefits, in terms of efforts, observed on Figure 12 are relatively small, we have to keep in mind that this process is to be repeated, for instance, on the five APIs definition presented on Figure 1 with, let us say 10 versions each. Since we want these API to be integrated with each other, it ends in the production of $(5 * 10)^4$ adapters. A potential extrapolation of our results would give an effort reduction of 87% by using our approach.

8.3 Runtime Overhead

The adapter generation process does not raise much runtime overhead for the translation execution. Indeed, the generated AspectJ only contains inter-type declarations to introduce new attributes and methods used in the visitor. The weaving between the aspects and the legacy code is performed at load-time using AspectJ 5 that does not introduce significant overhead. Still the Java behavior code generated from the Kermeta compiler could be improved, in particular the compilation of the lexical closure used in Kermeta in the free mapping are rather naive. Nevertheless, in studying existing translations in the legacy code developed in Java, we have also highlighted that some basic Object-Oriented principles are often violated. For example, the bad use of the visitor pattern introduces the use of lots of Runtime Type Information tests which also decreases the efficiency of the translator. For these three reasons, there is no significant waste or saving of performance during the translation execution, in the considered case study.

9. RELATED WORK

Software evolution and maintenance of legacy systems is a constant challenge for software engineers. Most research works focus on evolution and maintenance of one given legacy system. Various techniques such as Design Patterns [4] [3], and more recently models and program transformations [24, 21] have been used for this purpose. While these works aim at modifying the legacy for technical upgrade or even migration, we aim at connecting several existing legacies through the alignment of their API. We propose a Domain Specific Language to generate an adapter pattern for a set of classes. In this context, the constraints we consider are different from legacy evolution since the existing code must be kept unchanged. The process we propose for that is related to two main domains: model-driven engineering, for specifying the mapping at a high-level of abstraction; aspect-oriented programming for integrating the wrappers with the legacy code in a non-intrusive way.

The model-driven engineering community has developed several languages to specify transformations based on mappings between concepts such as QVT [5], ATL/AMW [8] or graph-based languages such as Viatra [22], VMTS [23] or AGG [1]. These approaches are oriented to meta-models correspondence and to transformation rules production in order to perform data transformation or migration. In comparison, the process we propose is specific to APIs translation and inter-operation. Our graphical representation of mappings is based on ideas taken from the work of Hausmann [11]. We kept the graphical representation of mappings and we use some concepts to include mappings within each other. However, as mentioned in previous section 5, we did not keep simple relations between mappings because these relations introduce more complexity in the mapping description activity that users could have difficulties to manage.

Even if several papers discuss the issue of evolving aspect-oriented applications [15], several works have shown the relevance of adopting Aspect-Oriented paradigm to work with legacy systems. Belapurkar [6] use aspect-oriented programming to comprehend and maintain complex legacy systems. It shows how AspectJ can be used to perform static or dynamic analysis of legacy code to evaluate the impact of an interface change, identify dead code, generate a dynamic call graph or evaluate the impact of exceptions. Ng *et al.* [16], discuss how simple yet effective AOP constructs can facilitate the process of program comprehension. Contrary to these work, we use AOP as a composition operator to enrich existing classes in the legacy without modifying their code.

Hannemann *et al.* have illustrated the relevance of AspectJ to implement GoF design patterns [10]. They demonstrate modularity improvements in 17 of 23 patterns. In our approach, we use AspectJ to integrate new methods and attributes that are necessary to implement the visitor pattern in existing legacy code. We extend the interface of the *Element* class via AspectJ's open class mechanism as suggested in [10]. Moreover we differ from the previous approach because we generate the AspectJ code from a domain specific language that declares the mapping between two APIs.

Currently, we use Inter-Type Definition of AspectJ mainly for modifying existing API without modifying their code. We could obtain the same results with other approaches. Scala [17] for example proposes a mixin-class composition that can be used to introduce new member definitions of a class. In association with its "implicit" mechanism that allows extension of existing classes through a lexically scoped implicit conversion, Scala can replace Java and AspectJ [19] as a back-end for generation. Another similar approach for the introduction is the use of Composition Filters [2] where filters are aspects that act as proxies to messages and impose additional functionalities. These functionalities are activated based on conditions specified in the filters.

In this trend of AspectJ generation, Meta-AspectJ [12] and XAspects [18], which advocate the use of AspectJ as a back-end language for aspect orientation, are similar with our approach. Nevertheless, the main goal is different. The value of Meta-AJ or XAspects is found in the reasons why neither AspectJ nor code-generation alone is sufficient. Meta-AJ provides syntactic features to deal with the dynamic creation of aspects, such as wildcard characters in names specifications. Using Meta-AJ, a user can leverage the full power of Java to create complex joinpoints that AspectJ does not support. We use AspectJ with load-time weaving to bypass the absence of open-classes [7] support in the legacy code.

10. CONCLUSION AND PERSPECTIVES

We have presented a process to semi-automatically produce adapters from existing APIs. Our approach is based on pooling existing techniques together (reverse-engineering, model-to-model transformation, code generation and aspect weaving) to alleviate tedious and error-prone developments. Although the specific domain of APIs interoperability looked first quite unsuitable for applying techniques such as Domain Specific Modeling (DSM) or Model-Driven Engineering (MDE), we successfully achieved to provide another way of producing interoperability adapters by combining model-related and code generation techniques.

Our process has been successfully applied on the example presented in Section 2, paving the way for large scale deployment on Thomson Extensible Management System for Digital TV.

This idea of smoothly combining models, aspects and generative techniques with legacy code goes actually well beyond the problem of API adaptation. Since very few developments start from scratch nowadays, the idea of using MDE to generate all the code of an application looks quite naive, and thus rightfully induces some skepticism in programmer's minds. A key issue of extending the usage of MDE beyond specialized domains then lies in its ability to

smoothly blend with legacy applications or components [13] as illustrated here. We hope that the approach presented in this paper could constitute a first step into reconciling modeling level approaches with programming level ones.

Acknowledgments

We would like to thank Nicolas Christian from Thomson Green Valley for providing us with details about the MTEP and XMS API specifications and for passing on his experience regarding former manual practices to develop adapters.

11. REFERENCES

[1] AGG. The attributed graph grammar (agg) system. http://user.cs.tu-berlin.de/ gragra/agg/.

[2] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[3] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 265–279, New York, NY, USA, 2005. ACM.

[4] S. Barkataki, S. Harte, and T. Dinh. Reengineering a legacy system using design patterns and ada-95 object-oriented features. In SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international conference on Ada, pages 148–152, New York, NY, USA, 1998. ACM.

[5] W. Bast, M. Belaunde, X. Blanc, K. Duddy, C. Griffin, S. Helsen, M. Lawley, M. Murphree, S. Reddy, S. Sendall, J. Steel, L. Tratt, R. Venkatesh, and D. Vojtisek. Mof qvt final adopted specification. OMG document ptc/05-11-01, October 2005 http://www.omg.org/docs/ptc/05-11-01.pdf.

[6] A. Belapurkar. Use aop to maintain legacy java applications. IBM developer work, (March 2004) http://www.ibm.com/developerworks/java/library/j-aopsc2.html.

[7] C. Clifton and G. T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In In OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 130–145, 2000.

[8] M. D. Del Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In SAC '07: Proceedings of the 2007 ACM symposium on Applied computing, pages 963–970, New York, NY, USA, 2007. ACM.

[9] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In Models in Software Engineering: Workshops and Symposia At MoDELS 2007, Nashville, Tn, Usa, pages 7–15. Reports and Revised Selected Papers, Septembre-October 2007.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 161–173, New York, NY, USA, 2002. ACM.

[11] J. H. Hausmann and S. Kent. Visualizing model mappings in uml. In SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization, pages 169–178, New York, NY, USA, 2003. ACM.

[12] S. S. Huang, D. Zook, and Y. Smaragdakis. Domain-specific languages and program generation with meta-aspectj. ACM Trans. Softw. Eng. Methodol., 18(2):1–32, 2008.

[13] A. Misra, J. Sztipanovits, G. Karsai, M. Moore, A. Ledeczi, and E. Long. Model-integrated computing and integration of globally distributed manufacturing enterprises: Issues and challenges. Engineering of Computer-Based Systems, IEEE International Conference on the, 0:225, 1999.

[14] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In S. K. L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264–278, Montego Bay, Jamaica, Oct. 2005. Springer.

[15] F. Munoz, B. Baudry, and O. Barais. Improving maintenance in aop through an interaction specification framework. In the proceedings of the 24th International conference on Software Maintentance, ICSM08, 2008.

[16] D. Ng, D. Kaeli, S. Kojarski, and D. Lorenz. Program comprehension using aspects. IEE Seminar Digests, 2004(902):89–96, 2004.

[17] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, P. Haller, S. Mcdirmid, S. Micheloud, N. Mihaylov, L. Spoon, and M. Zenger. A Tour of the Scala Programming Language, May 2007.

[18] M. Shonle, K. Lieberherr, and A. Shah. Xaspects: an extensible system for domain-specific aspect languages. In OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 28–37, New York, NY, USA, 2003. ACM.

[19] D. Spiewak and T. Zhao. Method proxy-based aop in scala. JOT: Journal of Object Technology, 8(7), 2009.

[20] S. A. Stelting and O. M.-V. Leeuwen. Applied Java Patterns. Prentice Hall Professional Technical Reference, 2001.

[21] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 295–312, New York, NY, USA, 2008. ACM.

[22] VIATRA. Viatra2. http://www.eclipse.org/gmt/VIATRA2/.

[23] VMTS. Visual modeling and transformation system (vmts). http://vmts.aut.bme.hu/.

[24] J. Zhang. Supporting software evolution through model-driven program transformation. In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 310–311, New York, NY, USA, 2004. ACM.

Author manuscript, published in "Software Tools for Technology Transfer (STTT) (2014)"

Noname manuscript No.
(will be inserted by the editor)

Generating Counterexamples of Model-based Software Product Lines

João Bosco Ferreira Filho · Olivier Barais · Mathieu Acher · Jérôme Le Noir · Axel Legay · Benoit Baudry

Received: date / Accepted: date

Abstract In a Model-based Software Product Line (MSPL), the variability of the domain is characterized in a variability model and the core artifacts are base models conforming to a modeling language (also called metamodel). A realization model connects the features of the variability model to the base model elements, triggering operations over these elements based on a configuration. The design space of an MSPL is extremely complex to manage for the engineer, since the number of variants may be exponential and the derived product models have to be conforming to numerous well-formedness and business rules. In this paper, the objective is to provide a way to generate MSPLs, called *counterexamples* (also called anti-patterns), that can produce invalid product models despite a valid configuration in the variability model. We describe the foundations and motivate the usefulness of counterexamples (e.g., inference of guidelines or domain-specific rules to avoid earlier the specification of incorrect mappings; testing oracles for increasing the robustness of derivation engines given a modeling language). We provide a generic process, based on the Common Variability Language (CVL) to randomly search the space of MSPLs for a specific modelling language. We develop LineGen a tool on top of CVL and modeling technologies to support the methodology and the process. LineGen targets different scenarios and is flexible to work either with just a domain metamodel as input or also with pre-defined variability models and base models. We validate the effectiveness of this process for three formalisms at different scales (up to 247 metaclasses and 684 rules). We also apply the approach in the context of a real industrial scenario involving a large-scale metamodel.

Keywords Software Product Lines · Model-based Engineering · Counterexamples

This work was developed in the VaryMDE project, a bilateral collaboration between the Diverse team at INRIA and the Thales Research & Technology. A preliminary version of this paper was published in the International Software Product Line Conference.

J. B. F. Filho · O. Barais · M. Acher · A. Legay
INRIA and IRISA, Université Rennes 1, France
E-mail: joao.ferreira_filho@inria.fr

Benoit Baudry
INRIA and SIMULA RESEARCH LAB, Rennes, France and Lysaker,Norway

Jérôme Le Noir
Thales Research & Technology, Palaiseau, France

1 Introduction

A *Software Product Line* (SPL) is a set of similar software products that share common features and assets in a particular domain [34]. Based on a desired set of features – usually documented in a variability model – the corresponding domain assets are combined during a so-called product derivation process. There can be different automation levels of product derivation, from manual development effort to more sophisticated technologies, including automated variant configuration and generation [3]. The challenge for practitioners is to develop and exploit what products have in common and manage what varies among them[37,10]. SPL engineering has emerged to address the problem [14,34] involving both the research community and the industry.

Model-based SPLs (MSPLs) have the same characteristics and objectives of an SPL, except that it extensively relies on models and automated model transformations. Models, as high-level specifications of a sys-

hal-01026581, version 1 - 22 Jul 2014

hal-01026581, version 1 - 22 Jul 2014

tem, are traditionally employed to automate the generation of products as well as their verifications [36]. The derivation of the customized models, corresponding to a final product, is achieved through a set of transformations.

1.1 MSPL in Industry and CVL

A variety of models may be used for different development activities and artefacts of an SPL – ranging from requirements, architectural models, source codes, certifications and tests to user interfaces. Likewise, different stakeholders can express their expertise through specific modeling languages (also called *metamodels*) and environments. It is an important requirement in large companies like Thales [43]. Many domains of expertise are indeed involved during the design, development and certification of systems. Stakeholders, whatever their roles or professions (requirement engineers, system engineers, software developers) in the organization, use a specific *language* to express his or her expertise. Stakeholders also use an associated and specific environment for elaborating and evolving the models, checking their complex properties, etc. As variability cross-cuts all development phases of an MSPL – from requirements to testing, stakeholders necessarily face the need to manage commonalities and variabilities within their models and throughout their specific modeling environments.

Numerous MSPL techniques have been proposed (e.g., see [34,32,29,15,13,18,45,42]). They usually consist in *i)* a variability model (e.g., a feature model or a decision model), *ii)* a model (e.g., a state machine, a class diagram) expressed in a specific modeling language (e.g., Unified Modeling Language (UML) [27]), and *iii)* a realization layer that maps and transforms variation points into model elements. Based on a selection of desired features in the variability model, a derivation engine can automatically synthesise customized models – each model corresponding to an individual product of the SPL. The *Common Variability Language (CVL)* [25] has recently emerged as an effort to standardize and promote MSPLs (see Section 2 for background information). For instance, Thales prototypes the use of CVL on dedicated domain-specific modeling languages for systems engineering.

1.2 Supporting the design of an MSPL

The *design space* (also called domain engineering) of an MSPL is extremely complex to manage for a developer. First, the number of possible products of an MSPL is exponential to the number of features or decisions expressed in the variability model. Second, the derived product models[1] have to be conformant to numerous well-formedness and business rules expressed in the modeling language (e.g., UML exhibits 684 validation rules in its EMF implementation). The number of derived models can be infinite while only part of the models are safe and conforming to numerous well-formed and business rules. Consequently, a developer has to understand the intrinsic properties of the modeling language when designing an MSPL. Last but not least, the two modeling spaces should be properly connected so that all valid combinations of features (configurations) lead to the derivation of a safe model. It is easy to forget a constraint between features in a variability model and allow a "valid" configuration despite the derivation of an unsafe product. It is also easy to specify a mapping that both delete and add the same model element for a given configuration. In the case of CVL, the realization model that connects a variability model and a set of design models, can be very expressive.

A one-size-fits-all support for designing MSPLs is unlikely, since models are conformant to their own well-formedness (syntactic) rules and domain-specific (semantic) rules. Each time a new modeling language is used for developing an MSPL, the realization layer should be revised accordingly. We observed this kind of situation in the context of prototyping the use of CVL with Thales. For instance, in [24], we expose different strategies to customize the derivation engine since the one provided by default in CVL does not suit the needs. Without adequate support, a developer of an MSPL is likely to introduce errors. The tooling support can provide different facilities: anti-patterns (counterexamples) to document what should be avoided during the design of an MSPL; domain-specific rules to avoid earlier the specification of incorrect mappings; examples to show possible correct MSPL, etc. Moreover, the support offered to domain experts should be ideally specific to a domain metamodel. Methodological support and guidelines are also needed to identify what constructs of a metamodel are likely to vary; to define an accurate realization model; or to develop specific derivation engines for a given modeling language.

[1] CVL uses the term materialization to refer to the derivation of a model. Also, a selected/unselected feature corresponds to a positively/negatively decided VSpec. We adopt the well-known vocabulary of SPLE for the sake of understandability.

hal-01026581, version 1 - 22 Jul 2014

1.3 Contributions

In this article, the objective is to provide a way to generate *counterexamples of MSPLs*, that is, examples of MSPLs that authorize the derivation of syntactically or semantically invalid product models despite a valid configuration in the variability model. These counterexamples aim at revealing errors or risks – either in the derivation engine or in the realization model – to stakeholders of MSPLs. On the one hand, counterexamples serve as testing "oracles" for increasing the robustness of checking mechanisms for the MSPL. Developers can use counterexamples to foresee boundary values and types of MSPLs that are likely to allow incorrect derivations. On the other hand, stakeholders may repeat the same kind of errors when specifying the mappings between a variability model and a base model. Counterexamples act as "antipatterns" that should avoid bad practices or decrease the amount of errors for a given modeling language.

We provide a systematic and automated process, based on CVL, to randomly search the space of MSPLs for a specific formalism (see Section 3). We develop a generic tool for supporting the methodology and approach initiated in [22]. The tool, called LineGen, aims to assist developers of MSPL and builders of MSPL tools by generating counterexamples of MSPLs (see Section 4). expressed in a given domain metamodel. LineGen relies on CVL and is flexible to target different scenarios. It can work either with just the domain metamodel as input or also with pre-defined variability models and base models. More details about LineGen can be found online: `https://code.google.com/p/linegen/wiki/LineGen`.

We validate the effectiveness of this process for three formalisms (UML, Ecore and a simple finite state machine) with different scales (up to 247 metaclasses and 684 rules) and different ways of expressing validation rules (see Section 5).

Another extension of our previous work [22] is the application of the approach in an industrial setting (see Section 6). We describe the rationale behind the introduction of LineGen and CVL at Thales; we also report on how we do scale for a very large metamodel with 20+ domain-specific modeling languages.

2 Background and Motivation

2.1 Model-based Software Product Lines

An SPL is a set of similar software products that share common features and assets in a particular domain. The process of constructing products from the SPL and domain assets is called *product derivation*. Depending on the form of implementation, there can be different automation levels of product derivation, from manual development effort to more sophisticated technology, including automated variant configuration and generation.

An MSPL has the same characteristics and objectives of an SPL, except that it extensively relies on *models*. In an MSPL, domain artefacts (requirements, tests, graphical interfaces, code) are represented as models conformant to a given modeling language, also called metamodel. (For instance, state machines can be used for specifying and testing the behavior of a system.) The goal of an MSPL is to derive customized models, corresponding to a final product, through a set of *automated transformations* [42,16].

Numerous approaches, being annotative, compositional or transformational, have been proposed to develop MSPLs (see Section 7 for more details). We will use the *Common Variability Language (CVL)* throughout the paper. We chose CVL because many of the MSPL approaches are actually amenable to this language (CVL is an effort involving both academic and industry partners to promote standardization for MSPLs).

2.2 Common Variability Language

In this section, we briefly present the main concepts of CVL and introduce some formal definitions that are useful for the remainder of this paper. CVL is a domain-independent language for specifying and resolving variability over any instance of any MOF[2]-compliant metamodel. The overall principle of CVL is close to many MSPL approaches: (i) A variability model formally represents features/decisions and their constraints, and provides a high-level description of the SPL (domain space); (ii) a mapping with a set of models is established and describes how to change or combine the models to realize specific features (solution space); (iii) realizations of the chosen features are then applied to the models to derive the final product model.

CVL offers different constructs to develop an MSPL, and they can be distinguished in three parts:

- **Variability Abstraction Model (VAM)** expresses the variability in terms of a tree-based structure. Inspired by feature and decision modeling approaches [17], the main concepts of the VAM are the variability specifications, called *VSpecs*. The *VSpecs* are nodes

[2] The Meta-Object Facility (MOF) is an OMG standard for modeling technologies. For instance, the Eclipse Modeling Framework is more or less aligned to OMG's MOF.

hal-01026581, version 1 - 22 Jul 2014

of the VAM and can be divided into three kinds (Choices, Variables, or Classifiers). In the remainder of the paper, we only use the *Choices VSpecs*, making the VAM structure as close as possible to a Boolean feature model – the variant of feature models among the simplest and most popular in use [8]. These *Choices* can be decided to yes or no (through *ChoiceResolution*) in the configuration process.

- **Base Models (BMs)** a set of models, each conforming to a domain-specific modeling language (e.g., UML). The conformance of a model to a modeling language depends both on well-formedness rules (syntactic rules) and business, domain-specific rules (semantic rules). The Object Constraint Language (OCL) is typically used for specifying the static semantics. In CVL, a base model plays the role of an asset in the classical sense of SPL engineering. These models are then customized to derive a complete product.
- **Variability Realization Model (VRM)** contains a set of Variation Points (VP). They specify how *VSpecs* (i.e., *Choices*) are realized in the base model(s). An SPL designer defines in the VRM what elements of the base models are removed, added, substituted, modified (or a combination of these operations, see below) given a selection or a deselection of a *Choice* in the VAM. But in the last iteration we could identify discrepancies. With respect to the variability model, we have found evidences that it is a tough taks to design it without leading to any wrong product models. It is also unfeasible to predict every possible configuration, once this number can reach exponential.

Using CVL, the decision of a *Choice* will typically specify whether a condition of a model element, or a set of model elements, will change after the derivation process or not. In this way, these choices must be linked to the model elements, and the links must explicitly express what changes are going to be performed. The aforementioned links compose the VRM, determining what will be executed by the **derivation engine**. Therefore, these links contain their own meaning. We consider that these links can express three different types of semantics:

- **Existence**. It is the kind of VP in charge of expressing whether an object (*ObjectExistence* variation point) or a link (*LinkExistence* variation point) exists or not in the derived model.
- **Substitution**. This kind of VP expresses a substitution of a model object by another (*ObjectSubstitution* variation point) or of a fragment of the model by another (*FragmentSubstitution*)
- **Value Assignment**. This type of VP expresses that a given value is assigned to a given slot in a base model element (*SlotAssignment* VP) or a given link is assigned to an object (*LinkAssignment* VP).

Using the models provided by CVL, one can completely express the variability over any MOF-compliant BM. In addition, it is possible to derive a family of models that will compose an MSPL. Therefore, it is possible to properly define an MSPL in terms of CVL (see Definition 1).

Definition 1 (Model-based SPL) *An MSPL* $= \langle CVL, \delta \rangle$ *is defined as follows:*

- *A* $CVL = \langle VAM, VRM, BMS \rangle$ *model is a 3-tuple such that:*
  - VAM *is a tree-based structure of* VSpecs. *We denote* $\mathcal{C}_{VAM}$ *the set of possible valid configurations for* VAM*;*
  - VRM *is a model containing the set of mapping relationships between the* VAM *and the* BM[3]*;*
  - $BMS = \{BM_1, BM_2, \ldots, BM_n\}$ *is a set of models, each conforming to a modeling language;*
- $\delta : CVL \times c \rightarrow DM$ *is a function that produces a derived model* DM *from a* CVL *model and a configuration* [4] $c \in \mathcal{C}_{VAM}$*. This function represents the derivation engine.*

2.3 Issues in Realizing Variability

We now introduce our running example to illustrate CVL and the issues raised when developing an MSPL.

Running Example. Let us consider the Finite-State Machine (FSM) modeling language. As shown in Figure 1, the FSM metamodel has three classes: *State*, *Transition*, and *FSM*. The metamodel defines some rules and constraints: a finite state machine has necessarily one initial state and a final state; a transition is necessarily associated to a state, etc. Some other rules may be expressed with OCL constraints (they are not in Figure 1 for conciseness), for example, to specify that there are no *State*s with the same name.

Using CVL and the metamodel of Figure 1, we can define a family of finite state machines. As shown in Figure 2, the VAM is composed by a set of *VSpecs*, while the VRM is a list of variation points, binding the VAM to the BM. The BM is a set of states and transitions conforming to the metamodel presented in Figure 1. The schematic representation of Figure 2 depicts a VAM (left-hand side) with 6 boolean choices (e.g., VS_5 and VS_6 are mutually exclusive) as well as a

[3] realization layer in the current CVL specification
[4] resolution model in CVL specification

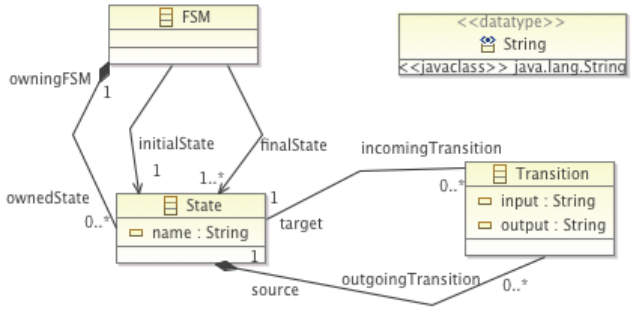


Fig. 1 FSM metamodel.

VRM that maps VS_3, VS_2, VS_5 and VS_6 to transitions or states of a base model denoted BM.

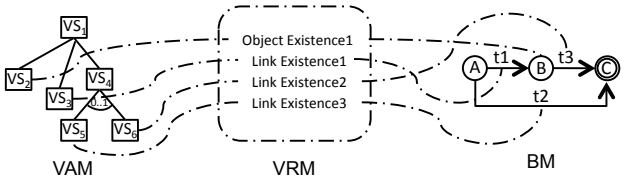


Fig. 2 CVL model over an FSM base model.

Considering the MSPL of Figure 2, it is actually possible to derive incorrect FSM models even starting from a valid BM and valid configurations of VAM. This is illustrated in Figure 3. Configuration 1 generates a correct FSM model, i.e., conforming to its metamodel. $Configuration$ 2 and $Configuration$ 3, despite being valid configurations of the VAM, lead to two unsafe products. Indeed, the FSM model generated from $Configuration$ 2 is not correct: according to the metamodel, an outgoing transition must have at least one target state, which does not hold for transition t1. In the case of $Configuration$ 3, the derived product model has the incoming transition t3 without a source state, which also is incorrect with respect to the metamodel.

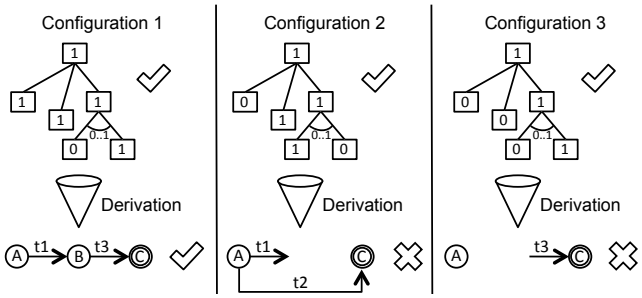


Fig. 3 Configuration and derivation of FSMs.

Even for a very simple MSPL, several unsafe product models can be derived in contradiction to the intention of an MSPL designer. In practice, specifying a correct MSPL is a daunting and error-prone activity due to the fact that the number of choices in the VAM, the number of classes and rules in the metamodel and the size of the VRM can be bigger.

The problem of safely configuring a feature or a decision model is now well understood [8]. Moreover, several techniques exist for checking the conformance of a model for a given modeling language. The connection of both parts (the VAM and the set of base models) and the management of the realization layer are still crucial issues [40,6,38,18,13].

3 Generating Counterexamples

We argue that the realization layer may concern at least two kinds of users:

- designers of MSPLs in charge of specifying the VAM, the BMs, as well as the relationships between the VAM and the BMs (VRM)(see CVL of Definition 1);
- developers of derivation engines in charge of automating the synthesis of model products based on a selection of features (*Choices*) (function δ of Definition 1);

Incorrect derivation engines or realization models may authorize the building of unsafe products. The majority of the existing work target scenarios in which an existing MSPL has been designed and seeks to first check its consistency, then to generate unsafe product models – pointing out errors in the MSPL. These techniques are extremely useful but assume that a generic derivation engine exists and is correct for the targeted modeling language – which is hardly conceivable in our case. Moreover, designers of MSPLs are likely to perform typical errors for a given modeling language (e.g., FSM).

3.1 Counterexamples to the Rescue

We precisely want to provide support to the two kinds of users in their activities. Specifically, we are interested on finding MSPLs that apparently would derive models that respect the domain modeling language, as they have a correct variability model and a conforming base model, but however, either their VRM or their derivation engine were incorrectly designed. Definition 2 formalizes this kind of MSPL as *counterexamples*.

Definition 2 (Counterexample of MSPL) *A counterexample CE is an $MSPL$ in which:*

- *CVL is well-formed;*

hal-01026581, version 1 - 22 Jul 2014

– *There exists at least one valid configuration in VAM:* $\mathcal{C}_{VAM} \neq \emptyset$*;*
– $\exists\ c \in \mathcal{C}_{VAM}, \delta\ (CVL,\ c\ , BM) =\ DM'$ *such that* DM' *does not conform to its modeling language.*

The expected benefits are as follows:

– SPL designers in charge of writing CVL models, can better understand the kinds of errors that should be avoided (Figure 3 gives two "antipatterns").
– developers of derivation engines can exploit counterexamples as testing oracles, anticipating the kinds of inputs that should be properly handled by their implementation. Furthermore, they can enrich the derivation engine with domain specific validation rules. In addition, specific error reports can be generated when an MSPL is incorrect, inspired by the catalogue of counterexamples.

3.2 Overview of the Generation

In order to systematically generate counterexamples of MSPLs, we have defined a set of activities that can be performed for this purpose. Figure 4 presents an overview of the process that generates a single counterexample, as well as the input and output for the different phases. We have divided the process into four phases, which are explained in details in the following subsections; the second and the third phases are part of the greater activity of generating a CVL model.

1. The first phase is the set up of the input that will be taken into account; different activities can be performed, depending on the input.
2. The second phase is the generation of a random variability model and of a valid random configuration.
3. The third phase is the generation of the relationships between the VAM and the base model elements, i.e., the variability model (VRM).
4. The fourth and last phase is to identify whether the generated model is a counterexample or not. In case it is not, we go back to the second step.

3.3 Set up input

Generally, companies that use or decide to set up a product line already have an initial set of core assets. In the case of MSPLs, if the models are not available, it is common to have the metamodel and the well-formedness rules of the modeling language. Considering this, the metamodel and the rules of the domain-specific modeling language are a starting point to generate a CVL model. Our approach is adaptable to work with both cases, whether the models are available or only their metamodel. In the case they are not available, we apply randomizations over the metamodel to create random models. These random instances populate the Base Model, and their correctness is checked against the metamodel and the well-formedness rules. If a created model is not correct, this instance is discarded. In the case of the FSM modeling language, the checked well-formedness rules are: if the initial state is different of the final, if the FSM is deterministic and if all the states are reachable. On the other hand, if we already have a set of models, we can use mutation operators to increase the number of samples, or just not modify the base models. Mutations operators are basic CRUD (Create, Read, Update, Delete) operations on the base model that are applied randomly.

3.4 Generate VAM and Resolution

For generating the VAM and the VRM, the following parameters are required:

– The maximum depth of the VAM *(MAX_DEPTH)* and the maximum number of children for each $VSpec$ *(MAX_CHILDREN)*.
– The percentage of $VSpecs$ that will be linked to variation points *(LINK_PERCENT)*. For example, in Figure 4, the VAM was generated with a percentage of 66%, as four out of six $VSpecs$ are linked to VPs.

Once the BM is established and the parameters have been set, we take them as input to start the generation of the CVL model. First, if the VAM is not provided by the user, we generate it, creating a root $VSpec$ and its children. The number of children is decided randomly, ranging from 0 to *MAX_CHILDREN*. The $VSpec$ creation is repeated for each generated child until the *(MAX_DEPTH)* is reached or there are no more $VSpecs$ with children. The only imposed generation is of the root node of the tree, after, it is a random decision between creating (or not) each child.

After generating the VAM, it is necessary to check its correctness, as we are not interested in wrong VAMs. For this reason, we translate the VAM to a language that can provide us a background for analysing it. The FAMILIAR language is executable and gives support to manipulate and reason about feature models [1] (we could also rely on existing frameworks like FaMa [8]). As stated in Section 2.2, the kinds of VAM we consider in this paper are amenable to boolean feature models supported by FAMILIAR. Using FAMILIAR, we check whether the variability model is valid or invalid. If it is an invalid model, we discard it and return to the VAM

hal-01026581, version 1 - 22 Jul 2014

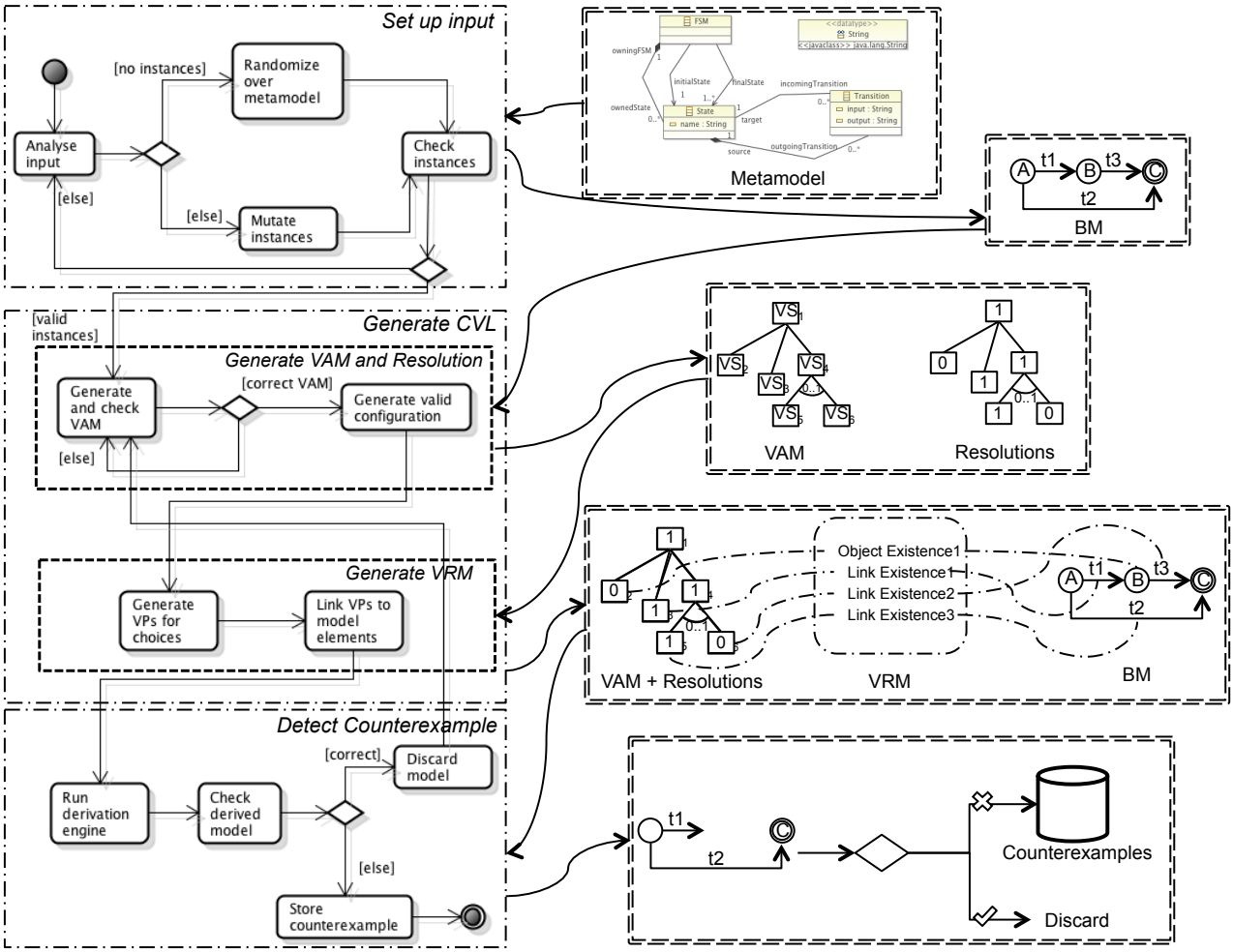


Fig. 4 Overview.

generation step. A resolution model is necessary in order to resolve the variability expressed in the VAM. To generate the configuration, we create the corresponding resolution CVL element for each $VSpec$. Meanwhile, random values (true or false) are set for each *ChoiceResolution* that has been created. We use standard satisfiability techniques to randomly generate a resolution, which is, by construction, a valid configuration of the VAM.

3.5 Generate VRM

Once we have a correct VAM and a correct BM, we can generate the VRM to link each other. To do this, we iterate over the set of choices in the VAM, deciding if the given choice is pointed or not by a Variation Point. This decision is done based on the *(LINK_PERCENT)* parameter. If the decision is true, we create the VP in the VRM. The type of the VP is also random. To finish the creation of the VP, we also randomize its target over the set of model elements of the BM. Naturally, we restrict the set of the randomization with respect to the kind of VP, e.g., a *LinkExistence* has a random target randomized over the subset of BM references. The VRM generation can also be independent, from existing VAMs and BMs, one could then explore the possibilities of relationships between them.

3.6 Detect Counterexample

Although Figure 4 describes the process of generating one single counterexample, we iterate the process to produce a set of counterexamples. For this reason, the first parameter to be taken into account is the stopping criteria. The stopping criteria can be specified in two different ways. The first one defines a target number of counterexamples, making the process repeat until this number is reached. The second one is to set an amount of time, stopping the process after it has elapsed.

hal-01026581, version 1 - 22 Jul 2014

hal-01026581, version 1 - 22 Jul 2014

After the aforementioned steps have been performed, we have a correct CVL model, composed by a correct VAM and a VRM created in conformance to the CVL metamodel. We also have a valid configuration c and a correct set of models composing the BM. The next step is to derive a product model using the CVL, c and the BM. If the derived model is incorrect, in other words, having δ $(CVL,\ c\ , BM)$ incorrect, we have found a counterexample as states the Definition 2, and consequently, we add it to the oracle. If the model is correct then we discard it and we come back to the generate VAM phase, synthesising a new entire CVL model.

The derivation engine is an algorithm that visits each of the variation points in the CVL model, executing them according to the resolution of the variability model. Our implementation of the CVL derivation engine follows the operational semantics of each variation point defined in the CVL specification (for further details, see the Annex A of the CVL revised submission provided in `http://www.omgwiki.org/variability`). To check if the derived model is correct, we relied on the EMF Diagnostician, using it as a black box to validate the conformance of the generated instance of the given metamodel.

As we will discuss in Section 4, these counterexamples can be helpful to the domain experts in charge of designing the CVL model or developing their derivation engines for their domain.

4 Tool Support

To support the process of generating counterexamples of MSPLs (exposed in the previous section), we developed a dedicated tool, called *LineGen*. Figure 5 gives an overview of the main features of LineGen. Depending on the inputs, the tool addresses different scenarios of counterexamples' generation – from the whole exploration of a modeling space (in the case only a metamodel is given) to the design of a specific MSPL (the variability model and the base model can be given by the user).

Specifically, the only mandatory input for LineGen is the metamodel of the base language. Additionally, the user can choose to provide existing base model and variability model; if this is the case, LineGen will not modify these models, setting them as immutable during the generation. To generate an MSPL example or counterexample, LineGen synthesizes a variability model, a configuration, a base model, and a set of realization relationships. LineGen calls the EMF's Diagnostician and checks the conformance of the base model with its input metamodel. After, LineGen checks the correctness of the variability model and the satisfiability of the configuration; to do so, it uses the reasoning engine within the FAMILIAR language. If they pass, LineGen carries on generating the realization relationships, finishing the CVL model.

After everything is generated, LineGen calls the CVL derivation engine, giving as input the generated CVL model (the triplet: variability model, realization model and base model) and a configuration. The goal of the call to the derivation engine is to determine whether the derived model is conforming to its modeling language. If it is, the CVL model given as input to the derivation engine is considered as an example of MSPL; otherwise it is considered as a counterexample.

We used different technologies as part of the LineGen implementation. As the user interface is an Eclipse 4 RCP application, it is written in Java. The core algorithms of the model generation parts are in Scala; we chose to use the same language in which we previously implemented the CVL derivation engine. We used the EMF API to manipulate and check the Ecore metamodels and model instances. To benefit from automatic analysis of the variability model, we translated the VAM to the FAMILIAR language [21].

Figure 5 shows the graphical user interface of LineGen. The user must load the Ecore metamodel of the modeling language to be able to perform the generation steps (see ①). Once the metamodel has been successfully loaded—the Console (see ⑥) shows whether LineGen successfully completed an operation or not—it is possible to generate a base model by pressing the Generate BM button (see ②); a file named *BaseModel* is created with the chosen extension. The *Max Many* field should be set to limit the number of instances of a given model element.

The same process applies to the VAM generation (see ③ in Figure 5). The user specifies the maximum depth of the VAM, as well as the maximum number of children per feature. After pressing the Generate VAM button, LineGen creates a CVL model with just the VAM part defined. In the VRM tab, the user can define the percentage of features linked to a variation point in the VRM (see ④).

In the Counterexamples and Examples tabs (see ⑤), the user can start the generation process that will randomly search for examples or counterexamples of MSPLs. If the user chooses to use the Load Existing Base Model or Load Existing VAM tabs, LineGen uses the loaded models without modifying them and just generates VRM models. The field Number of Counterexamples determines when LineGen has to stop the search. The Console tab also provides exception messages, in case an unexpected error occurs. More details

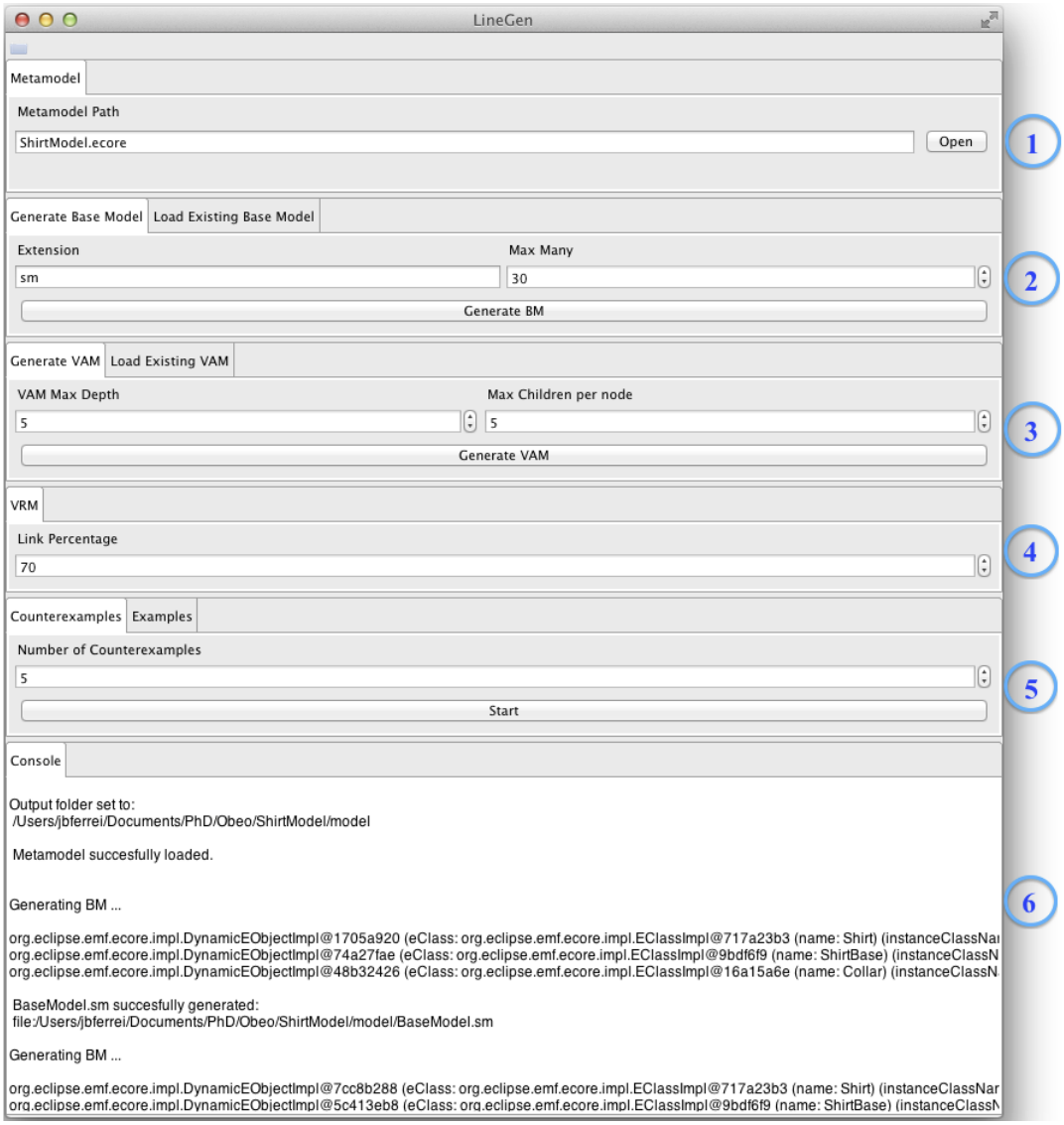


Fig. 5 LineGen user interface

about LineGen can be found online: `https://code.google.com/p/linegen/wiki/LineGen`.

5 Evaluation

The goal of this evaluation is to verify the applicability and effectiveness of the proposed approach, as well as to assess important properties of the generated counterexamples. Regarding the effectiveness, we formulated the following question:

- RQ1. Can the approach generate counterexamples in a reasonable amount of time?

Then we seek to answer questions about the properties of the generated counterexamples, such as:

- RQ2. Does the number of counterexamples increase in a more complex domain?
- RQ3. With respect to the metamodel or the OCL rules, what errors are the most common in the counterexamples?
- RQ4. Is it possible to prevent the generation of counterexamples by the designer?

5.1 RQ1. (Applicability and Effectiveness)

Answering this question will allow us to know if the approach can actually generate counterexamples and how long it takes to generate a range of counterexamples.

Objects of Study. To answer RQ1, we need to apply the proposed approach to specific scenarios and verify if it effectively produces counterexamples. As a

hal-01026581, version 1 - 22 Jul 2014

hal-01026581, version 1 - 22 Jul 2014

first scenario, we use the FSM modeling language that was presented in previous sections. As second and more complex scenario, we use the Ecore modeling language. We provide the corresponding metamodel and validation rules as input for both scenarios. As previously mentioned, the FSM metamodel has 3 classes and 4 rules, while the Ecore metamodel has 20 metaclasses, 33 datatypes and 91 validation rules. We set up the parameters equally for both scenarios: the stopping criteria is set to the number of 100 counterexamples, the *MAX_DEPTH* is set to 5, the *MAX_CHILDREN* is set to 10 and the *LINK_PERCENT* is set to 30%.

Experimental Setup. Once the parameters and the input are ready, we start the automatic generation of the counterexamples. The generation was performed in a machine with a 2nd Generation Intel Core I7 processor - Extreme Edition and 16GB of 1333MHz RAM memory, running under a linux 64bit with a 3.8.0 kernel, Scala 2.9.3 and an oracle Java Runtime Environment 7.

Experimental Results. The times are shown in Figure 6, ranging from 0 to 12625 seconds. For both FSM and Ecore, we could successfully find and generate counterexamples in a reasonable time. The time for generating 10 counterexamples for the Ecore-based MSPL was approximately 15 minutes, which is acceptable, considering the complexity of the Ecore metamodel. Thus, as the target number of counterexample increases, we can confirm a linear growth of the time. The linear trendlines are a good fit to the obtained time values, with R^2 values close to 1. Each time value is an average of 10 executions, this was done to minimize the random effect.

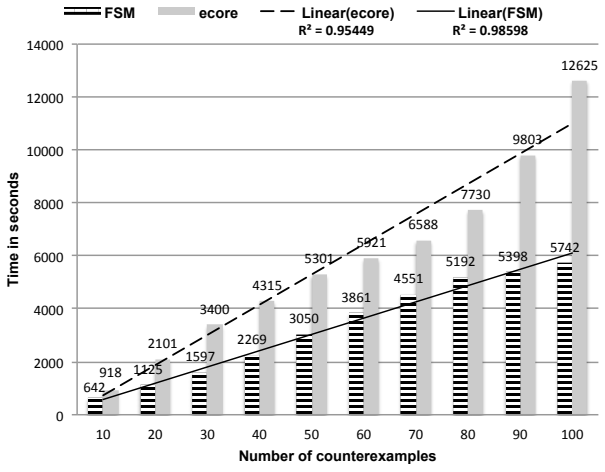


Fig. 6 Counterexamples for FSM and Ecore.

5.2 RQ2. (Counterexamples vs Domain Complexity)

This research question aims at analysing the consequences of applying the approach in a more complex domain. Answering this question helps whether and to which extent it is more likely to design counterexamples (i.e., unsafe MSPLs) when the domain becomes more complex or not.

Objects of Study. To address RQ2, we compared the ratio between the number of invalid DMs and valid DMs. We made this comparison with three different modeling languages: FSM, Ecore (with the Eclipse Modeling Framework implementation) and UML (with the Eclipse UML2 project implementation. We classified these modeling languages in the following increasing sequence of complexity: FSM $<$ Ecore $<$ UML. Indeed, the FSM metamodel contains only 3 metaclasses 1 datatype and 4 validation rules. The Ecore metamodel contains 20 metaclasses, 33 datatypes and 91 validation rules. Finally, the UML contains 247 metaclasses, 17 datatypes and 684 validation rules.

Experimental Setup. For each modeling language, we applied our approach to obtain 100 counter examples, using the same parameters of the first experiment, and we collect the number of correct DMs we obtain. The evaluation was performed on the same computer of the previous experiment. For generating valid UML model, we do not create UML models from scratch, but we mutate existing UML models. We chose the footnote referred set of UML models to create the BM[5].

Experimental Results. The experiment resulted in the generation of 469 correct DMs for 100 counterexamples for FSM, 292 correct DMs for 100 counterexamples for Ecore and 52 correct DMs for 100 counter examples for UML[6]. We can therefore verify the ratio of incorrect per correct derived models. In the case of FSM, the ratio is 1 incorrect DM to 5 correct DMs, while in the case of Ecore, this ratio is 1 to 3, and for UML the ratio is 1 to 0,5. These results provide evidence that, as the domain modeling language becomes more complex, the chance to get a correct DM becomes lower. In a sense, it confirms the relevance of our procedure for generating counterexamples. More importantly, the practical consequence is that the designer is likely to produce much more unsafe MSPLs when the targeted modeling language is complex.

[5] http://goo.gl/kC0sx

[6] Source code for the experiment is available at http://goo.gl/PgkrL

hal-01026581, version 1 - 22 Jul 2014

5.3 RQ3. (Nature of the errors)

The purpose here is to evaluate whether the errors are a violation to the structural properties of the metamodel or to the validation rules (i.e., OCL rules). Answering this question can help to understand which part of the modeling language is more likely to reveal more errors. Hence, we conducted the following experiment to investigate the research question.

Objects of Study. To identify the nature of the errors in the counterexamples, we used the generation of the 100 counterexamples for the three modeling languages that were previously used to answer RQ2. Our object of study is the quantity of counterexamples with errors violating the metamodel or the OCL rules.

Experimental Setup. For each modeling language, we applied our approach to obtain 100 counterexamples under the same parameters, and then we identify in which part of the modeling language definition is the error of the DM. The evaluation was performed using the same computer of the previous experiment.

Experimental Results. For the FSM language, among the 100 counterexamples, we generate 10 models that do not conform to the metamodel and 90 models that violate one of the validation rules. For the Ecore modeling language, among the 100 counter examples, we generate 64 models that do not conform to the metamodel and we generate 36 models that violate one of the validation rules. For the UML modeling language, among the 100 counter examples, we generate 22 models that do not conform to the metamodel and we generate 78 models that violate one of the validation rules.

We now correlate these numbers with the properties of the modeling language. FSM contains only three structural rules (i.e., a state-machine must contain at least one state, one initial state and at least one final state). Most of the errors are the validation rules that are violated. Ecore contains much more structural rules (mainly lower case constraints for cardinality). Therefore lots of errors come from structural inconsistencies. Finally UML contains so many validation rules that it is unfeasible to create a valid UML model randomly. (That is why we used *mutation* from a set of valid UML models.) For this case we obtained much more DMs that violate validation rules expressed in OCL.

Yet, it is hard to draw definitive conclusions on whether structural or validation rules expressed in OCL participate the most in generating incorrect MSPLs. The results indicate that the kind of errors that are the most common in the counterexamples depend mainly on the domain modeling language (Ecore vs UML). It is well known, for instance, that some OCL rules can be refactored as structural constraints in the metamodel. In a sense, it partly confirms – in the context of CVL – some of the results exposed in [9] showing there exists different "styles" of expressing business or domain-specific rules within a metamodel.

5.4 RQ4. (Antipattern Detection)

The purpose of RQ4 is to evaluate the feasibility of expressing validation rules on the triplet VAM, BM, VRM to decrease the risk of creating invalid DMs from a valid CVL model and a correct BM, being $\mathcal{C}$ the set of possible valid configurations for a valid VAM. This question helps to know if it is possible for a domain designer to detect early "bad" CVL models (acting as "antipatterns") for a given domain.

Objects of Study. To evaluate this research question, we created two validation rules to detect antipattern for the FSM modeling language. Rule 1 prevents a substitution between a final state and an initial state, and vice versa. Rule 2 constrains the fact of having an object existence that targets the initial state of an FSM. These rules have been implemented in Scala and can be written in few lines using an OCL writing style, as shown in Listing 1.

Listing 1 Antipattern rules for FSM

```
1  def checkVRM(f:FSM,vrm: VPackage):Boolean = {
2      vrm.asInstanceOf[VPackage].
           getPackageElement().foreach(e=> {
3  /*Rule 1: Replacing a final state by an
       initial one, and vice versa, is
4  forbidden.*/
5          if (e.isInstanceOf[ObjectSubstitution
               ]){
6          var p = e.asInstanceOf[
               ObjectSubstitution].
               getPlacementObject().getReference()
7          var p1 = e.asInstanceOf[
               ObjectSubstitution].
               getReplacementObject().getReference
               ()
8          if ((f.getFinalState().contains(p) && f
               .getInitialState().equals(p1))|| (f
               .getFinalState().contains(p1) && f.
               getInitialState().equals(p)))
               return false;
9          }
10 /*Rule 2: Pointing an ObjectExistence to an
       initial state is forbidden.*/
11         else if (e.isInstanceOf[
               ObjectExistence]){
12           e.asInstanceOf[ObjectExistence].
                 getOptionalObject().foreach(p=>
                 {if (f.getInitialState().equals(
                 p.getReference())) return false
                 ;})}})
13     return true}
```

Experimental Setup. For the FSM modeling language, we applied our approach to obtain 100 counterexamples and we compare the number of valid DMs we obtain either checking the antipatterns rules or not. The evaluation was performed on the same computer that the previous experiment, as well as with the same parameters.

Experimental Results. The experimental results show that we generate 1860 correct DMs for 100 counterexample for FSM when the antipattern rules for CVL are activated, against 469 correct DMs for 100 counter examples for FSM when the CVL validation rules for CVL are not activated. For this domain, writing only 2 rules on the triplet of VAM, VRM, BM allowed us to decrease 4 times the risk of generating an invalid DM. Therefore, it is feasible to detect identified antipatterns using our approach, writing validation rules that detect *a priori* and therefore earlier these errors.

5.5 Discussion

Besides the checking operations, the time results presented in Figure 6 are mainly dependent on the following factors:

1. The time to generate a correct set of models to compose the BM;
2. The time to generate a correct VAM;
3. The time to generate a VRM;

These three factors are resulting from the generality and the full automation of our approach that does not require any input models. The approach gives the ability of finding possible design errors without having yet designed the MSPL. This allows users to explore the design space of an MSPL, given a modeling language – this is the main scenario we initially target. However, it is possible to *predefine some inputs*. It could enhance the scalability of our generative process, since there is no need to spend time in generating these inputs. It may be the case when a designer of an MSPL already has an established BM. Another possible situation is when the VAM has been previously designed, as it is often one of the starting points of an MSPL. Therefore, we can claim that the conducted experiment address the *worst case* input for our approach. Consequently, our approach is sufficiently generic, as it does not assume that it is always the case of having a VAM or the BM as input. In addition, because it is fully automated, the approach does not demand a great effort to be used. Another benefit of predefining some inputs is that we could address other scenarios, like the debugging of an existing MSPL or the definition of various realization models given predefined BMs and VAMs.

By definition, an MSPL is a complex structure, composed by different connected models. This characteristic makes hard to design a correct MSPL, as errors can occur in any design phase. Given this great proneness to error, it is relevant to discuss the causes and to reason where is the lack of safety. For this purpose, we can analyse and give a rationale about two questions:

1. How a VAM and its analysis tools check and prevent configurations that result in incorrect DMs?
2. Is the fact of a derivation operator generate an incorrect DM fault of the own derivation operator (derivation engine) or is it fault of how it was invoked (realization model)?

Regarding the first question, it seems unfeasible to have a generic checker that, for any domain, could detect whether a configuration derives or not an incorrect model. It is rather needed to customize a derivation engine and/or a consistency checker (e.g., a simulator [44]) that takes into account the syntactic and semantic rules of the domain. Likewise, faulty configurations, currently not supported by the MSPL, could be better identified and located. From this aspect, counterexamples can help to devise such specific simulators and oracles. For the second question, we can argue that there is a trade-off between the expressiveness of the realization model and the safeness of the derivation. On the one hand, if more restrictions are applied to the derivation engine, we limit what could be generated. Also, a realization design can be wrong in one domain, but correct in another. On the other hand, if the derivation engine is not customized to address the specific meanings of a modeling language, then it is necessary to have checking mechanisms for the VRM that takes into account the syntax and semantics of the domain. More practical investigations are needed to determine when to customize the derivation engine or when to develop specific checking rules for the VRM. Counterexamples can be used for implementing both solutions.

6 Approach in an Industrial Case

In the last session, we evaluated our approach against well-known modelling languages; we could verify that it produces counterexamples in a reasonable time and we could also assess properties of the counterexamples. In this section, we present how the approach performs facing an industrial case. First, we describe the company's scenario; second, we report on how we could successfully apply the approach on it; and finally, we reproduce the applicability and effectiveness experiments done in the RQ1 of the evaluation.

6.1 Thales Scenario

Thales is a large company involved with different industry sectors (aerospace, space, defence and transportation areas, etc.); they produce software intensive systems, using model-based technologies, and they seek to

hal-01026581, version 1 - 22 Jul 2014

hal-01026581, version 1 - 22 Jul 2014

evolve towards a product line approach. Thales already has a well-established and functional model-based method for developing their systems and software, the ARCADIA, however they seek to leverage this development from single software to families of software, maintaining their safety and quality standard [23].

The ARCADIA method is a viewpoint-based architectural description, defining 5 different abstraction levels of a system, following the ISO/IEC 42010, Systems and Software Engineering - Architecture Description [30]. Thales' engineers use numerous domain specific modeling languages to develop integrated sets of systems according to ARCADIA. These languages are built within a set of dedicated representations to analyze specific problems. The language workbench provides a set of customizable and highly dynamic representations working seamlessly together on top of models. These representations can be combined and customized according to the concept of Viewpoints. Views, dedicated to a specific Viewpoint, can adapt both their display and behavior depending on the model state and on the current concern. The same information can also be simultaneously represented through diagram, table or tree editors.

These languages are defined as a set of 20 metamodels with about 400 metaclasses and about 200 validation rules; they model the ARCADIA method in an eclipse-based environment. Besides, this workbench is extensible and new languages can be defined to design specific viewpoints of a system. Therefore, leveraging product line engineering for each of these languages and domains is very expensive and error-prone; it has to be supported by automated tools.

Several stakeholders have to work during the design process on the tool chain:

- Product-line engineers who have to identify the commonalities and the variants and in charge of designing the VAM and the VRM.
- Product engineers who have to create specific products, focusing on creating valid products regarding a set of requirements.
- DSL designers who are in charge of creating or extending existing DSLs (base metamodel). They define where and how we can put variability within (at the M2 level) the architecture and the derivation semantics [24]

The use of the proposed counter example framework aims at easing the correct cooperation between these stakeholders. It is used to provide a pragmatic approach to guide these stakeholders to design CVL model that provides only valid products.

6.2 Approach Application and Results

We applied the approach to the Thales' representative sample model of weather balloons; this base model has 2079 model elements and 563Kb and, despite of being one single subdomain, it can serve as a pilot application for other similar areas of the organization. The set of metamodels and validation rules of ARCADIA are considered as input to the approach. In contrast of what we did to evaluate the approach in a generic way (generating everything else besides the metamodel), we could simplify the generation because Thales provided a variability model and the aforementioned preliminary base model, narrowing down the problem space. Therefore, we fixed the VAM and the BM, randomizing only over the configurations of their variability model and generating the set of variation points to compose the realization model. However, it was necessary to adapt the implementation to meet some technical requirements from Thales for loading and saving the models.

Reproducing the same experimental setup of RQ1, we performed 10 rounds and measured the average time for generating 100 counterexamples. The results in seconds are shown in Figure 7. We could verify that in a situation where the VAM and the BM are provided, it is around 27 times faster to generate the same amount of counterexamples, and the curve still behaves linearly.

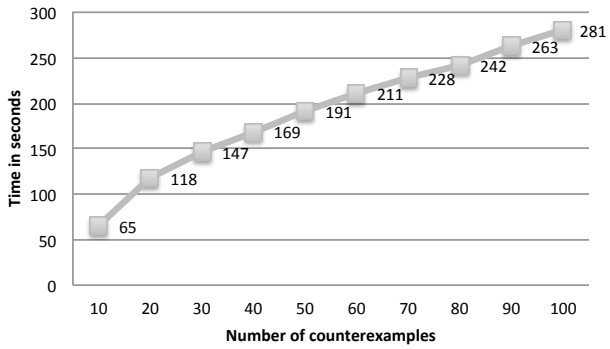


Fig. 7 Counterexamples for ARCADIA sample model.

About 30% of the models generated for this domain were wrong, meaning that, in average, if we randomly define realization relationships among the features and the model elements of this domain, almost one third can result on counterexamples. Another interesting result is the fact that only one OCL rule added to the VRM can remove 50% of the counterexamples (Forbidden an object existence on a specific kind of model element "*EventSendCallAction*") and 80% of the counterexamples can be removed in writing 8 basic OCL rules. With this example, we can show that the generation of coun-

hal-01026581, version 1 - 22 Jul 2014

terexamples from a reference model can help to detect some anti-patterns that can be easily constrained and detected for a particular domain. The result is the improvement of the use of CVL in this industrial context an early detection of CVL model that capture invalid products.

7 Related Work

MSPLs. Different variability modeling approaches have been proposed. *Annotative* approaches derive concrete product models by activating or removing parts of the model. Variant annotations define these parts with the help of, for example, UML stereotypes [45] or presence conditions [13,18,15]. *Compositional* approaches associate model fragments with product features that are then composed for a particular configuration (i.e., combination of features). For instance, Perrouin *et al.* offer means to automatically compose modeling assets based on a selection of desired features [32]. Apel *et al.* propose to revisit superimposition technique and analyze its feasibility as a model composition technique [4]. Dhungana *et al.* provide support to semi-automatically merge model fragments into complete product line models [19]. Annotative and compositional approaches have both pros and cons. Voelter and Groher illustrated how negative (i.e., annotative) and *positive* (i.e., compositional) variability [42] can be combined. *Delta modeling* [35,11] promotes a modular approach to develop MSPL. The deltas are defined in separate models and a core model is transformed to a new variant by applying a set of deltas.

The variability realization layer of CVL, as exposed in Section 2.2, provides the means to support annotative, compositional or transformational approaches [39, 28]. Therefore we believe our work is applicable to a wide range of existing MSPL approaches.

Verification of SPLs. Some techniques specifically address the problem of verifying SPL or MSPL[5]. The objective is usually to guarantee the *safe composition* of an SPL, that is, all products of an SPL should be "safe" (syntactically or semantically). In [40], Batory *et al.* proposed reasoning techniques to guarantee that all programs in an SPL are type safe: i.e., absent of references to undefined elements (such as classes, methods, and variables). At the modeling level, Czarnecki *et al.* presented an automated verification procedure for ensuring that no ill-structured template instance (i.e., a derived model) will be generated from a correct configuration [18]. In [13,12], the authors developed efficient model checking techniques to exhaustively verify a family of transition systems against temporal properties. Asirelli *et al.* proposed a framework for formally reasoning about modal transition systems with variability [6]. In [2], Alfeérez *et al.* applied VCC4RE (for Variability Consistency Checker for Requirements) to verify the relationships between a feature model and a set of use scenarios. Zhang *et al.* [44] developed a simulator for deriving product models as well as a consistency checker. Svendsen *et al.* present an approach for automatically generating a testing oracle for train stations expressed in CVL [38].

Some of this work generate counterexamples when the property of safe composition is violated, typically for presenting to a developer an error in the specification of an SPL. In our approach, the goal is not to produce unsafe products of an *existing* MSPL, but to generate unsafe MSPLs. We do not assume variability models, models or configurations as inputs and the approach is fully automated. We thus target scenarios that go beyond debugging an existing MSPL. Our objective is rather to *prevent* the unsafe specification of realization models, i.e., generated counterexamples act here as "anti-patterns" that should prevent practitioners in specifying unsafe MSPLs. Another important difference is that verification techniques previously described assume that the derivation engine is correct. In our context, we cannot formulate the same hypothesis and have rather the crucial needs to implement new and robust derivation engines – each time a new modeling language is used in the MSPL. We provide quantitative evidence that the specificity of the modeling language should be taken into account. The generation of counterexamples aims at producing testing "oracles" and guide developers when building a derivation engine.

Techniques for combinatorial interaction testing of feature models (the VAM part of CVL) [31,33,26] have been proposed. As future work we plan to consider their use as part of our generation process.

Classification. In [41], Thüm *et al.* present a classification and survey of analysis strategies for SPLs. According to their classification, our work can be seen as an incremental product-based approach analysis, due to the fact that we produce and explore sample products. A limitation that follows every product-based approach is that it takes exponential effort to guarantee that the SPL model is safe. Given the design space complexity of a modeling language, it is unlikely to generate every possible counterexamples. We thus rely on randomization to synthesize a finite set of counterexamples.

Besides, we recognize as a limitation the fact that, in our experiments, we did not consider coverage criteria of counterexamples. It is hard to specify such coverage criteria for any domain-specific modeling language. For example, it is hard to synthesize a set of base models that covers any constructs offered by a language. Our

approach has the merit of being agnostic of a domain and leaves to the user the choice of how exhaustive he/she wants to search the problem space. Considering this scenario, our target is not to directly provide mechanisms to verify the safety of an existing SPL (i.e., ensuring there is no unsafe product). As motivated by our industrial case study with Thales, we rather aim to help the engineers in charge of building mechanisms for MSPLs.

Verification and debugging of models. Numerous techniques have been proposed for debugging or verifying consistency of models or model transformations (e.g., [7,20]). These works do not address specific issues of MSPL engineering, especially those related to the realization layer.

8 Conclusions and Future Work

Because of the combinatorial explosion of possible derived variants, the great variety and complexity of its models, correctly designing a Model-based Software Product Line (MSPL) has proved to be challenging. It is easy for a developer to specify an incorrect set of mappings between the features/decisions and the modeling assets, thus authorizing the derivation of unsafe product models in the MSPL. In this continuation paper, we have presented a systematic and fully automated approach to explore the design space of an MSPL. The main objective of the approach was to generate counterexamples of MSPLs, i.e., MSPLs that can produce invalid product models. This kind of MSPL can be used to test derivation engines or provide examples of invalid VRMs, which could serve as a basis to establish antipatterns for developers.

For this purpose, we have formalized the concepts of an MSPL, based on the Common Variability Language (CVL), as well as the concept of a counterexample. We explained in details each step of our generative approach and illustrated it with a running example. The tool LineGen, built on top of CVL and modeling technologies, supports the generative process. It enables practitioners to explore the whole design space of a given modeling language but also to focus on a specific MSPL with a pre-defined variability and base models. We performed experiments to assess the applicability and effectiveness of the tool-supported approach. The conducted experiments allowed us to evaluate the approach when applied to different modeling languages, at different scales of complexity. We could successfully generate counterexamples for each modeling language in a reasonable amount of time, which could be drastically reduced when the approach received additional input. In addition, we explored the natures of errors found in the counterexamples and our ability to detect antipatterns. We also reported on our experience when instantiating the approach and LineGen in an industrial context.

As future work, we seek to automate as much as possible the safe construction of an MSPL, using the counterexamples and examples as material to design an MSPL. In particular, we want to combine machine-learning techniques to automatically classify the counterexamples w.r.t. the kinds of error. Our hope is to provide an efficient approach to synthesize domain-specific rules that can prevent earlier the detected anti-patterns.

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Familiar: A domain-specific language for large scale management of feature models. Science of Computer Programming (SCP) Special issue on programming languages p. 22 (2013). DOI http://dx.doi.org/10.1016/j.scico.2012.12.004
2. Alférez, M., Lopez-Herrejon, R.E., Moreira, A., Amaral, V., Egyed, A.: Supporting consistency checking between features and software product line use scenarios. In: K. Schmid (ed.) ICSR, *Lecture Notes in Computer Science*, vol. 6727, pp. 20–35. Springer (2011)
3. Apel, S., Batory, D., Kstner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer (2013)
4. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In: R.F. Paige (ed.) ICMT, *Lecture Notes in Computer Science*, vol. 5563, pp. 4–19. Springer (2009)
5. Apel, S., Rhein, A.v., Wendler, P., Grosslinger, A., Beyer, D.: Strategies for product-line verification: Case studies and experiments. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 482–491. IEEE Press, Piscataway, NJ, USA (2013). URL `http://dl.acm.org/citation.cfm?id=2486788.2486852`
6. Asirelli, P., ter Beek, M.H., Gnesi, S., Fantechi, A.: Formal description of variability in product families. In: E.S. de Almeida, T. Kishi, C. Schwanninger, I. John, K. Schmid (eds.) SPLC, pp. 130–139. IEEE (2011)
7. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM **53**(6), 139–143 (2010)
8. Benavides, D., Segura, S., Ruiz-cort, A.: Automated Analysis of Feature Models 20 Years Later : A Literature Review. Information Systems **35**(6) (2010)
9. Cadavid, J.J., Baudry, B., Sahraoui, H.A.: Searching the boundaries of a modeling space to test metamodels. In: ICST, pp. 131–140 (2012)
10. Chen, L., Babar, M.A., Ali, N.: Variability management in software product lines: a systematic review. In: SPLC'09, pp. 81–90 (2009)
11. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proceedings of the 9th GPCE'10 conference, GPCE '10, pp. 13–22. ACM, New York, NY, USA (2010). DOI 10.1145/1868294.1868298. URL `http://doi.acm.org/10.1145/1868294.1868298`
12. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE'11, pp. 321–330. ACM (2011)

hal-01026581, version 1 - 22 Jul 2014

hal-01026581, version 1 - 22 Jul 2014

13. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE'10, pp. 335–344. ACM (2010)
14. Clements, P., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional (2001)
15. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, *LNCS*, vol. 3676, pp. 422–437 (2005)
16. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering, GPCE '02, pp. 156–172. Springer-Verlag, London, UK (2002)
17. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions: a comparison of variability modeling approaches. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems, VaMoS '12, pp. 173–182. ACM, New York, NY, USA (2012). DOI 10.1145/2110147.2110167. URL `http://doi.acm.org/10.1145/2110147.2110167`
18. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE'06, pp. 211–220. ACM (2006). DOI http://doi.acm.org.gate6.inist.fr/10.1145/1173706.1173738
19. Dhungana, D., Grünbacher, P., Rabiser, R., Neumayer, T.: Structuring the modeling space and supporting evolution in software product line engineering. Journal of Systems and Software **83**(7), 1108–1122 (2010)
20. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Software Eng. **37**(2), 188–204 (2011)
21. FAMILIAR: FeAture Model scrIpt Language for manIpulation and Automatic Reasonning: `http://nyx.unice.fr/projects/familiar/`
22. Filho, J.B.F., Barais, O., Acher, M., Baudry, B., Noir, J.L.: Generating counterexamples of model-based software product lines: an exploratory study. In: T. Kishi, S. Jarzabek, S. Gnesi (eds.) SPLC, pp. 72–81. ACM (2013)
23. Filho, J.B.F., Barais, O., Baudry, B., Le Noir, J.: Leveraging variability modeling for multi-dimensional model-driven software product lines. In: Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on, pp. 5–8 (2012). DOI 10.1109/PLEASE.2012.6229774
24. Filho, J.B.F., Barais, O., Le Noir, J., Jézéquel, J.M.: Customizing the common variability language semantics for your domain models. In: Proceedings of the VARiability for You Workshop, VARY '12, pp. 3–8. ACM, New York, NY, USA (2012). DOI 10.1145/2425415.2425417. URL `http://doi.acm.org/10.1145/2425415.2425417`
25. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Svendsen, A., Zhang, X.: Standardizing Variability - Challenges and Solutions. In: SDL Forum, pp. 233–246 (2011)
26. Gotlieb, A., Hervieu, A., Baudry, B.: Minimum pairwise coverage using constraint programming techniques. In: G. Antoniol, A. Bertolino, Y. Labiche (eds.) ICST, pp. 773–774. IEEE (2012)
27. Group, O.M.: OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Tech. rep. (2007). URL `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`
28. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference, SPLC '08, pp. 139–148. IEEE Computer Society, Washington, DC, USA (2008). DOI 10.1109/SPLC.2008.25. URL `http://dx.doi.org/10.1109/SPLC.2008.25`
29. Heidenreich, F., Sanchez, P., Santos, J., Zschaler, S., Alferez, M., Araujo, J., Fuentes, L., amd Ana Moreira, U.K., Rashid, A.: Relating feature models to other models of a software product line: A comparative study of featuremapper and vml*. Transactions on Aspect-Oriented Software Development VII, Special Issue on A Common Case Study for Aspect-Oriented Modeling **6210**, 69–114 (2010)
30. ISO: International organization for standardization: Iso/iec fcd 42010: Systems and software engineering - architecture description (2010)
31. Johansen, M.F., Haugen, Ø., Fleurey, F.: An algorithm for generating t-wise covering arrays from large feature models. In: E.S. de Almeida, C. Schwanninger, D. Benavides (eds.) SPLC (1), pp. 46–55. ACM (2012)
32. Perrouin, G., Klein, J., Guelfi, N., Jézéquel, J.M.: Reconciling automation and flexibility in product derivation. In: SPLC'08, pp. 339–348. IEEE (2008). DOI http://dx.doi.org/10.1109/SPLC.2008.38
33. Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., Traon, Y.L.: Pairwise testing for software product lines: comparison of two approaches. Software Quality Journal **20**(3-4), 605–643 (2012)
34. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
35. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proceedings of the 14th international conference on Software product lines: going beyond, SPLC'10, pp. 77–91. Springer-Verlag, Berlin, Heidelberg (2010). URL `http://dl.acm.org/citation.cfm?id=1885639.1885647`
36. Schmidt, D.C.: Model-Driven Engineering. IEEE Computer **39**(2) (2006)
37. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques: Research articles. Softw. Pract. Exper. **35**(8), 705–754 (2005). DOI http://dx.doi.org/10.1002/spe.v35:8
38. Svendsen, A., Haugen, Ø., Møller-Pedersen, B.: Specifying a testing oracle for train stations - going beyond with product line technology. In: J. Kienzle (ed.) MoDELS Workshops, *LNCS*, vol. 7167, pp. 187–201. Springer (2011)
39. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K.: Developing a software product line for train control: A case study of cvl. In: J. Bosch, J. Lee (eds.) SPLC, *Lecture Notes in Computer Science*, vol. 6287, pp. 106–120. Springer (2010)
40. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe composition of product lines. In: GPCE '07, pp. 95–104. ACM, New York, NY, USA (2007). DOI http://doi.acm.org.gate6.inist.fr/10.1145/1289971.1289989
41. Thüm, T., Apel, S., Kästner, C., Schaefer, I., Saake, G.: A classification and survey of analysis strategies for software product lines. In: Computing Surveys, 2014. To appear; accepted 2014-01-30. ACM (2014)
42. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software develop-

ment. In: SPLC'07, pp. 233–242. IEEE (2007). DOI http://dx.doi.org/10.1109/SPLC.2007.28
43. Voirin, J.L.: Method & tools to secure and support collaborative architecting of constrained systems. In: 18^{th} International Symposium of the INCOSE. International Council on Systems Engineering, Utrecht, Netherlands (2008)
44. Zhang, X., Møller-Pedersen, B.: Towards correct product derivation in model-driven product lines. In: Ø. Haugen, R. Reed, R. Gotzhein (eds.) SAM, *Lecture Notes in Computer Science*, vol. 7744, pp. 179–197. Springer (2012)
45. Ziadi, T., Jézéquel, J.M.: Software product line engineering with the uml: Deriving products. In: T. Käkölä, J.C. Dueñas (eds.) Software Product Lines, pp. 557–588. Springer (2006)

hal-01026581, version 1 - 22 Jul 2014

Author manuscript, published in "Models 2012 (2012)"

hal-00714558, version 1 - 5 Jul 2012

An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements

François Fouquet[1], Grégory Nain[2], Brice Morin[3], Erwan Daubert[1], Olivier Barais[1] Noël Plouzeau[1], and Jean-Marc Jézéquel[1]

[1] University of Rennes 1, IRISA, INRIA Centre Rennes
Campus de Beaulieu, 35042 Rennes, France
{Firstname.Name}@inria.fr
[2] SnT - University of Luxembourg
Luxembourg, Luxembourg
{gregory.nain}@uni.lu
[3] SINTEF
Oslo, Norway
{Brice.Morin}@sintef.no

Abstract. Models@Runtime aims at taming the complexity of software dynamic adaptation by pushing further the idea of reflection and considering the reflection layer as a first-class modeling space. A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework. EMF provides facilities for building DSLs and tools based on a structured data model, with tight integration with the Eclipse IDE. EMF has rapidly become the defacto standard in the MDE community and has also been adopted for building Models@Runtime platforms. For example, Frascati (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities. This paper highlights specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative implementation to meet these requirements.

Keywords: Model@Runtime, EMF, adaptation

1 Introduction

The emergence of new classes of systems that are complex, inevitably distributed, and that operate in heterogeneous and rapidly changing environments raise new challenges for the Software Engineering community [3]. Examples of such applications include those from crisis management, health-care and smart grids. These applications can be deployed on top of a distributed infrastructure that goes from micro-controller to the Cloud. These systems must be adaptable, flexible, reconfigurable and, increasingly, self-managing [9]. Such characteristics make systems more prone to failure when executing and thus the development and

hal-00714558, version 1 - 5 Jul 2012

study of appropriate mechanisms for continuous design and runtime validation and monitoring are needed. In the Model-Driven Software Development area, research effort has focused primarily on using models at design, implementation, and deployment stages of development. This work has been highly productive with several techniques now entering the commercialization phase. The use of model-driven techniques for validating and monitoring run-time behavior can also yield significant benefits. A key benefit is that models can be used to provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns such as verification and monitoring. Then, Models@Runtime [2] denotes model-driven approaches aiming at taming the complexity of software and system dynamic adaptation. It basically pushes the idea of reflection [11] one step further by considering the reflection layer as a real model: "*something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality*" [14], which enables the continuous design of complex, adaptive systems.

A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework (EMF) [4]. For example, Frascati [16] (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities and its use to support Models@Runtime reaches some limitations. This paper elicits specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative modelling framework implementation to meet these requirements.

The outline of this paper is the following. Section 2 briefly presents the Models@Runtime paradigm and its requirements. An overview of EMF benefits and its limitations regarding its use at runtime are given by Section 3. The contribution of this paper, the Kevoree Modeling Framework(KMF), is described in Section 4. Section 4.2 gives an evaluation of our alternative implementation in comparison to EMF. This contribution is discussed *w.r.t.* related work in Section 5. Section 6 concludes on about this work and presents future work.

2 Models@Runtime Requirements

The Models@Runtime paradigm promises a new approach to MDE, by fading the boundary between design-time (the typical phase where MDE is employed) and runtime. More precisely, the goal of Models@Runtime is to enable the continuous design, evolution, verification of eternal running software systems [2]. A typical usage of Models@Runtime is to manage the complexity of dynamic adaptation or verification in complex, **distributed** and **heterogeneous** systems, by offering a more abstract and safer abstraction layer on top of the running system than reflection. Heterogeneity and distribution creates specific requirements for Models@Runtime infrastructure. (i) The overhead inevitably induced by this

[4] http://www.eclipse.org/emf/

hal-00714558, version 1 - 5 Jul 2012

advanced reflection layer should not prevent smaller (*i.e.* resource constrained) devices to benefit from the advantages of Models@Runtime (e.g. Java Embedded, Android,...). Modeling framework and all its needed dependencies must be compatible with such devices in terms of memory footprint. (ii) The use of models to drive the running configuration of a software system should enable required features of a distributed reflection layer such as efficient (un)-marshalling, efficient model cloning and model thread safety access.

2.1 Reduced Memory Footprints

The memory footprint of a Models@Runtime engine basically determines the types of nodes able to run this engine. The more demanding is the Models@Runtime engine in terms of memory, the more difficult it is to deploy it on the smallest devices (e.g. Android phones, gateways with low power CPUs), and the more centralized should the adaptation/verification be. Lazy loading technique can be used to virtually reduce the memory overhead by not loading unused model elements. In this case, only a few large devices would be able to reason and make decisions for all the smaller devices. This would reduce the reliability of the overall adaptation and verification process: if the large devices fail, the overall system cannot safely adapt anymore. Moreover, model exchanges for the synchronization of the system in this strategy would dramatically increase network load.

2.2 Dependencies

The number and size of dependencies is also an important criteria. Each device must provision all the dependencies needed by the modeling framework to run a Models@Runtime based distributed application. As these applications are based on a structured data model, this data model should not generate useless dependencies. Heavy dependencies would indeed increase the time needed to initialize a node or update it when new versions of those third parties are available.

2.3 Thread Safety

A Models@Runtime is generally used in highly concurrent environment. For instance, different probes integrated in a device update a context model. This model is then used for triggering the adaptation reasoning process. This context model should enable safe and consistent read and write for the reasoners to take accurate decisions. The Models@Runtime infrastructure must ensure that the multiple threads of your application can access and modify the models without worrying about the concurrent access details. In particular, it should be possible to navigate in parallel the collections defined in the model to implement fast, yet safe, validation or reasoning algorithms on multi-core/thread nodes.

hal-00714558, version 1 - 5 Jul 2012

2.4 Efficient Model (Un)Marshalling and cloning

A device should be able to locally clone its own model for verification or reasoning purposes so that it can reason on a fully independent and safe representation of itself, which can later on be re-synchronized with the current model. Also, devices should be offered efficient means to communicate their Models@Runtime to neighbors so that collective decisions can be made. The Models@Runtime infrastructure must thus provide efficient model cloning and (un)marshalling capabilities.

2.5 Connecting Model@Runtime to classical design tools

This requirement is directly bound to the first goal of fading the boundary between design-time and runtime. A Models@Runtime infrastructure must provide a transparent compatibility with design environments. For example, a graphical simulator used for the design of finite state machines (FSM) should be pluggable on an application that keeps FSM at runtime and serve as a debugger or a monitor of the running system [1,7].

3 EMF Benefits and Limitations

A natural way to implement a Models@Runtime platform is to rely on tools and techniques well established in the MDE community, and in particular, the *de facto* EMF standard. This section provides a brief overview of EMF and then discusses the suitability of this modelling framework with respect to the requirements identified in the previous section.

3.1 EMF Overview

EMF is an eMOF implementation and code generation facility for building tools and other applications based on a structured data model. From a model specification, EMF provides tools and runtime support to create Domain Specific Language (DSL) on top of the Eclipse platform. Most important, EMF provides the foundation for interoperability with other EMF-based tools and applications using a default serialization strategy based on XMI. Consequently, EMF has been used to implement a large set of tools and thus, evolved into an efficient Java implementation of a core subset of the MOF API.

3.2 Advantages.

As a first real benefit, EMF provides a transparent compatibility of the Models@Runtime infrastructure with several design environments. All the tools built with frameworks such as Xtext [10,4], EMFText [8], GMF [15] or ObeoDesigner [5]

[5] http://www.obeodesigner.com/

hal-00714558, version 1 - 5 Jul 2012

can be directly plugged on the Models@Runtime infrastructure to monitor the running system. The generated code is clean and provides an embedded visitor pattern and an observer pattern [6]. EMF also provides an XMI marshaller and unmarshaller that can be used to easily share models. Finally EMF offers lazy loadings of resources allowing the loading of single model elements on demand and caching them softly in an application.

3.3 Limitations.

To highlights the limitations of EMF, we will use the following experiment based on a simple Finite State Machine (FSM) metamodel with four meta-classes (FSM, State, Transition and Action in Fig. 1). A FSM contains the States of the machine and references to initial, current and final states. Each State contains its outgoing Transitions, and Transitions can contain Actions. From this tiny example, we discuss thread safety and dependencies, and we evaluate the memory footprint, as well as model (un)marshaling and cloning.

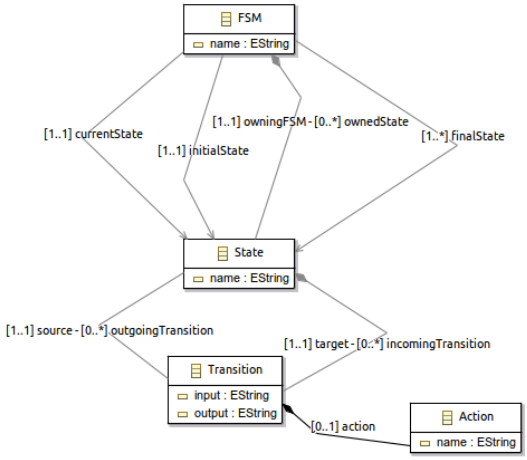


Fig. 1. Finite State Machine Metamodel used for Experiments

Large dependency set. Figure 2 shows the plugin/bundle dependencies for each new EMF generated code. By analyzing these dependencies one can see that the generated code is tightly coupled to the Eclipse environment and to the Equinox runtime (Equinox is the version of OSGi by the Eclipse foundation). Although this is not problematic when the data model is integrated as an Eclipse plugin (with all dependencies imposed by the Eclipse environment); these dependencies are more difficult to resolve and provision when this metamodel is used outside Eclipse, *i.e.* in a standalone context.

hal-00714558, version 1 - 5 Jul 2012

For the simple FSM metamodel, a standalone JAR executable outside of the Eclipse tool (a Java archive that including all dependencies) has a size of **15 MB** for only **55 KB** generated files. This footprint is rather difficult to reduce with tools like ProGuard[6], since it contains a large number of reflexive calls, which could potentially and implicitly affect any code in these 15 MB.

The large number and size of dependencies is one of the main limitations of EMF when the model must be embedded at runtime.

Static registries and multi-class loader incompatibility. Many runtime for dynamic architecture (e.g. OSGi, Frascati, or Kevoree) need to use their own class loader to properly manage and improve dynamic class loading. Consequently, the second limitation comes from the use of static registries in EMF, that leads to incompatibilities with runtime using multi-class loaders.

Lack of thread safe access to the models. EMF does not provide thread safe accesses to the models [7]. This requirement is important for a Models@Runtime infrastructure, because the support for dynamic and distributed architectures requires concurrent access to models.

Cloning overhead. Another limitation is the large memory footprint of marshaling, unmarshaling and cloning in the EMF implementations. To measure this limitation, we programmatically created a model with 100,000 State instances, with a transition between each state and an action for each Transaction. The results for EMF are the following.
On a Dell Precision E6400 with a 2.5 GHz iCore I7 and 16 GB of memory, the model creation lasts 376 ms, its marshaling to a file lasts 7021 ms and uses 104 MB of heap memory. The cloning using `EcoreUtil` lasts 3588 ms, and loading the model from a file lasts 5868 ms[8].

3.4 Synthesis.

Table 1 summarizes the advantages and limitations for the usage of EMF as a foundation for a Models@Runtime infrastructure. For each criterion, we put a $\vee$ when EMF provides advantages, $\times$ when we see limitations of using EMF for building Models@Runtime infrastructure, $\sim$ when we see possible improvements.

[6] ProGuard is a code shrinker (among other features not relevant for this paper) for Java bytecode: http://proguard.sourceforge.net/
[7] http://wiki.eclipse.org/EMF/FAQ#Is_EMF_thread-safe.3F
[8] This experiment can be downloaded http://goo.gl/CyLLC

hal-00714558, version 1 - 5 Jul 2012

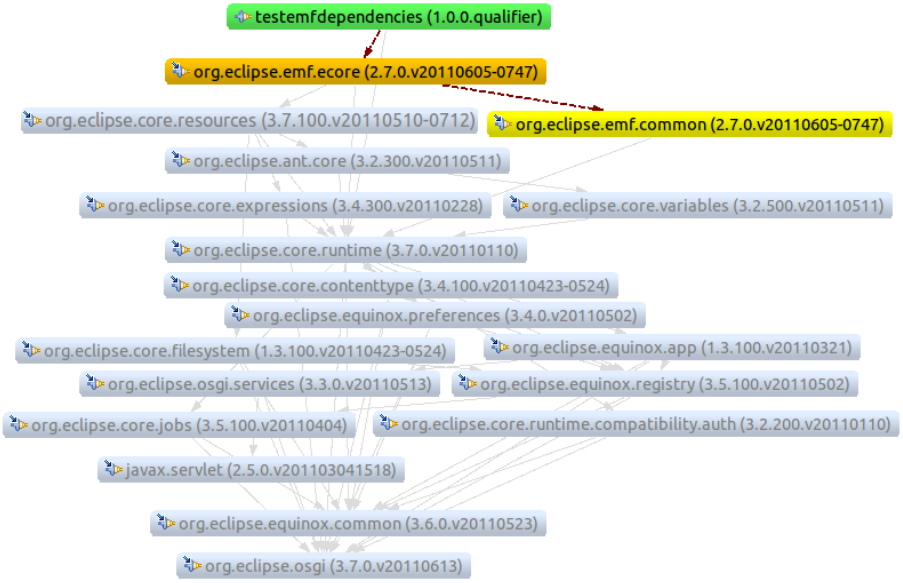


Fig. 2. Dependencies for each new metamodel generated code

Memory footprints	×
Lazy loading	∨
Dependencies	×
Thread safety	×
Efficient model (un)marshalling and cloning	∼
Connecting design tools	∨

Table 1. EMF features compared with Models@Runtime requirements

4 Kevoree Modeling Framework

KMF, or Kevoree Modeling Framework [9], is our alternative realization of EMF, which was formerly developed as part of our Kevoree Models@Runtime engine. This section presents the design choices we made to support a generic and efficient Models@Runtime infrastructure compatible with EMF. The general idea of KMF is threefold:

1. KMF aims at keeping the compatibility with EMF to guarantee the compatibility with design environment and the marshalling and unmarchalling of models.

[9] https://github.com/dukeboard/kevoree-modeling-framework

hal-00714558, version 1 - 5 Jul 2012

2. KMF aims at leveraging the powerful features provided by modern programming languages (here, Scala) [13] to provide a proper design to handle models.
3. KMF aims at providing a generic Models@Runtime infrastructure to ease the heterogeneity and the distribution management.

4.1 Model handling

Regarding the Table 1, KMF provides the same features than EMF for code generation facilities and models (un)marshalling. All the generated artefacts are written in Scala. Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented (Traits) and functional languages, and provides bytecode compatibility with Java [13]. Scala uses type inference to combine static safety with the concise syntax of dynamically typed languages. The Scala features particularly relevant for KMF are the following: ByteCode compatibility with Java libraries, concept of traits, concept of *Option*, XML embedded, immutable List, concept of closure and efficiency.

Domain classes are generated as a set of Scala traits to ease the support of multiple inheritance and meta-model extension [5]. Traits are seen from Java Code as a Type. They can only be initialized through the generated Factory (as in EMF). Note that the generated Traits do not inherit from EMF `EObject` and that all references are initialized. In particular, collections are initialized and references to single objects with a lower bound of 0 rely on Options. These Scala options are a neater way to deal with null pointers[10]. Indeed, Option does not save the developer from ever having null, but that developer can only get null when he wants it. If it is semantically impossible for a value to be null, the type checker enforces it. The getters on collection use immutable lists. The generated code provides helpers to add and remove model elements on collections, and it also provides specific methods to ease mixed Java/Scala development. XML template are directly embedded in the Scala generated code and type checked by the Scala type checker. Fig. 3 shows an excerpt of the Scala traits generated for the domain model meta-classes. It shows the use of immutable list and Option for 0..n reference and 0..1 reference respectively.

4.2 Memory footprint

To limit the dependencies, we decided to restrict the inheritance relationships in our generated code only to generated classes and to classes from the Java and the Scala frameworks. In this way dependencies are limited to the Java and Scala frameworks. A standalone JAR for the same metamodel in KMF has a size of 7 MB. After applying ProGuard, we obtain a JAR of 1.7MB. Indeed, the Scala dependencies load many packages that are not used by KMF, such as Scala-Swing, Scala-actors, etc.

[10] http://www.scala-lang.org/api/current/scala/Option.html

hal-00714558, version 1 - 5 Jul 2012

```
trait Transition extends FsmSampleContainer {
    private var input : java.lang.String = ""

    private var output : java.lang.String = ""

    private var source : fsmSample.State = _

    private var target : fsmSample.State = _

    private var action : Option[fsmSample.Action] = None
    ...
}


trait FSM extends FsmSampleContainer {

    // 0..n reference
    private var ownedState : scala.collection.mutable.ListBuffer[
        fsmSample.State] = scala.collection.mutable.ListBuffer[fsmSample
        .State]()
    ....
    // 0..n reference method helpers
    def getOwnedState : List[fsmSample.State] = {
        ownedState.toList
    }
    def getOwnedStateForJ : java.util.List[fsmSample.State] = {
        import scala.collection.JavaConversions._
        ownedState
    }

    def setOwnedState(ownedState : List[fsmSample.State] ) {
        this.ownedState.clear()
        this.ownedState.insertAll(0,ownedState)
        ownedState.foreach{e=>e.setEContainer(this,Some(()=>{this.
            removeOwnedState(e)}))}

    }

    def addOwnedState(ownedState : fsmSample.State) {
        ownedState.setEContainer(this,Some(()=>{this.removeOwnedState(
            ownedState)}))
        this.ownedState.append(ownedState)
    }

    def addAllOwnedState(ownedState : List[fsmSample.State]) {
        ownedState.foreach{ elem => addOwnedState(elem)}
    }

    def removeOwnedState(ownedState : fsmSample.State) {
        if(this.ownedState.size != 0 ) {
            this.ownedState.remove(this.ownedState.indexOf(ownedState))
            ownedState.setEContainer(null,None)
        }
    }

    def removeAllOwnedState() {
        this.ownedState.foreach{ elem => removeOwnedState(elem)}
    }

    def getClonelazy(subResult : java.util.IdentityHashMap[Object,Object
        ]): Unit = {
      ...
    }
}
```

Fig. 3. Excerpt of generated Scala code for domain meta-classes

hal-00714558, version 1 - 5 Jul 2012

Consequently, KMF has successfully been used on top of Dalvik [11], Avian [12], JamVM[13] or JavaSE for embedded Oracle Virtual Machine [14].

4.3 Multi-thread access

Model@Runtime serves as a common software reflection layer concurrently exploited by many processes; protection against such accesses may be coarse or fine grain.

At fine grain the model essentially needs to be read concurrently while allowing modifications. The KMF generated code realizes such protection by internally using mutable collections (for performance reasons) but only exposing cloned immutable list to outside via its public API. As a result processes can navigate the cloned list while others perform CRUD operations on it. Each process needs to actively ask a new cloned version to access to the modifications. Moreover, the mutator methods (setter) can be protected behind synchronized blocks.

At coarse grain the model representation is entirely hidden behind a safe model care tracker as in the Memento pattern [6]. This safe model care tracker systemically clones the model on *get* operations and keeps a master representation. This structure is particularly useful to keep an history of model representation at runtime. KMF can optionally generate such structure using Scala actors to protect concurrent access (get / put) to model care tracker.

4.4 Loader, Serializer and Cloner

When working with models, two tasks are essential and used before and after each action on a model, namely marshalling and unmarchalling.
Where EMF offers a generic loader we propose to use the generation phase to also generate a specific loader for each meta-model.
The EMF generic loader takes a model to load and its meta-model as parameters and intensively uses reflection mechanisms to perform the loading task. If this kind of mechanism allows creating a single loader, its usage is not efficient. The generated KMF code then provides meta-model specific loader, saver and cloner to improve their efficiency. We use the XML API which is part of the Scala standard library to parse and print XMI representations of object models, with no need for extra dependencies, and because it is efficient.
The loading and cloning are performed in two phases. The first phase consists in traversing the models for creating the objects in the order they are found in the XMI file. The last step links the objects together according to the references previously cached. Currently, the Maven plugin we propose is only able to generate

[11] http://www.dalvikvm.com/
[12] http://oss.readytalk.com/avian/
[13] http://jamvm.sourceforge.net/
[14] http://www.oracle.com/technetwork/java/embedded/downloads/javase/index.html

hal-00714558, version 1 - 5 Jul 2012

loaders and serializers for the XMI file format. EMF compatibility is obtained through the XMI file format. A direct API compatibility can be performed when EMF will separate the Ecore interfaces and Ecore implementation in different bundles to avoid useless dependencies.

4.5 Experiment and synthesis

	EMF	*KMF*	comparison
Model creation	376 ms	313 ms	1.2 times faster
Model clone	3588 ms	398 ms	9 times faster
Model save	7021 ms	2630 ms	2.66 times faster
Memory footprint	104MB of heap memory	61MB of heap memory	1.70 times lighter

Table 2. EMF and KMF efficiency

Besides, the memory footprints used to store, load, save or clone a model has decreased compared to the reference EMF implementation. To measure the memory footprints, (un)marshalling and cloning, we do the same experiment. We programmatically create models with 100 000 States with a transition between each state and an action in each Transaction.
The results for KMF are the following. On a Dell Precision E6400 with an Intel 2.5GHz iCore I7 CPU and 16GB of RAM, it takes 313ms to create the models, 2630 ms to save it in a file, 61 Mbytes of Heap memory, 398ms to clone and 3000s to load the model from a file[15]. Table 2 highlights the quantitative performance comparison results between EMF and KMF.

Table 3 provides the qualitative comparison results between EMF and KMF.

	EMF	*KMF*
Memory footprints	$\times$(104MB)	$\vee$(61MB)
Dependencies	$\times$ (15MB)	$\vee$ (Scala standard library) (1.7MB)
Lazy Loading	$\vee$	$\vee$ (Proxy support)
Thread safety	$\times$	$\vee$ (immutable lists, no registry)
Efficient model (un)marshalling and cloning	$\sim$	$\vee$ (see Table 2)
Design tools compatibility	$\vee$	$\vee$ (through XMI compatibility)

Table 3. EMF and KMF regarding models@runtime requirements

[15] This experiment can be download http://goo.gl/9Huwa (for eclipse project) or http://goo.gl/0sWRo (for the maven project)

hal-00714558, version 1 - 5 Jul 2012

5 Discussion

5.1 Refactoring impact

The first major consequence of removing the runtime dependencies with EMF is that all the methods defined in `EObject` are now unavailable. This could have a significant impact on the existing code that uses these methods. In order to limit the refactoring impact of this removal, we re-implemented the `eContainer` mechanism of EMF in our generated code.

Another important design choice we made for KMF was to use Scala as the default language for the generation and use of the code. Java has also been considered as a language since Scala code is fully compatible with Java. However, Scala code is not always friendly to use from a Java program[16]. To ease the use of KMF in a Java environment, we also provide a standard Java API, which in particular exposes Java lists, by duplicating some methods that are suffixed with "4J". That requires all model navigation-related code to be rewrote to use these new methods. The dual strategy could of course have been implemented: generating Java code and exposing in addition a Scala API.

The main rationale behind the choice of Scala was that the Scala standard library provides many facilities that are useful for Models(@runtime). In particular, the systematic introduction of Scala `Option` for each optional (*i.e.* having lower bounds equals to 0) attribute or reference implies to explicitly test if the element is defined or not, in a neater way than `if (myRef == null)` in Java, and in a safer way than a `NullPointerException` popping at runtime. Here again, this mechanism enforces developers to consider the optional aspect of these elements and avoids lots of null-checks, but requires a deep refactoring.

5.2 Limitations

KMF has formerly been developed and tested using the Kevoree metamodel originally designed with EMF tools. This first step allowed for setting up the basis for model, loader and serializer generation. The use of a fairly different metamodel (from Kermeta [12]) highlighted some missing features in the generation process, and strongly helped in improving KMF. However, KMF still has some limitations.
For instance, reverse relations have already be flagged as missing in the generated code.

EMF allows model elements to have some relations with elements from other metamodels (references, attribute types, inheritance, etc). This mechanism has been partially realized on a concrete use case, but it still need improvements and implementation discussions.

[16] To give an idea, a Scala list built by concatenating the empty list (Nil) and the element 1 would be written `1::nil` in Scala. The construction of the same Scala list in Java would yield `$colon$colon$.MODULE$.apply((Integer) 1, nil);`

hal-00714558, version 1 - 5 Jul 2012

Moreover, the generation of loaders and serializers relies on containment relations between model elements, and it requires a root container element. Until now, we considered metamodels that have only one single model element as root of the containment tree, but this is not the general case. Indeed, all model elements must have a container, but not necessarily under a single root for the metamodel. There could be several containment roots (and several containment trees) in a metamodel, with references to each other, but also across different metamodels. Generators of loaders and serializers are not ready to accept such kind of metamodels, but meeting this requirement is already considered as future work.

KMF addresses performances issues of models in memory (heap). Complementary approaches like CDO addresses the management of models in persistence memory (databases). The CDO (Connected Data Objects) Model Repository [17] is a distributed shared model framework for EMF models and metamodels. CDO has a 3-tier architecture supporting EMF-based client applications, featuring a central model repository server and leveraging different types of pluggable data storage back-ends like relational databases, object databases and file systems. The default client/server communication protocol is implemented with the Net4j Signalling Platform. This solution offers tools to easily do collaborative work and history management. However, it uses EMF as a local representation. Consequently it inherits a part of the drawbacks coming from EMF (e.g. dependencies for client side, memory footprints). Moreover the use of external server to manage history is not useful for pervasive systems that may have a sporadic network and even if it can embed server side on client, the overhead of the dependencies is not suitable for lightweight systems.

6 Conclusion

This position paper has discussed the needs for adapting the *de facto* standard in the MDE community, *i.e.* the Eclipse Modelling Framework (EMF), for a more dynamic usage of models in the context of Models@Runtime. After highlighting requirements related to Models@Runtime, this paper has presented an initial adaptation of EMF, named Kevoree Modelling Framework (KMF), implemented in Scala and generating code for this language. Even if KMF only supports XMI serialisation, it provides a significant speedup on model creation, model (un)marshalling and model cloning. It also has a lighter memory footprint than the reference implementation, and its runtime dependencies are limited to the Java and Scala libraries, whereas the EMF generated code has tight dependencies to Eclipse and Equinox. This significantly hinders the reusability of the EMF code outside Eclipse, while KMF code can run Eclipse-free on various Java virtual machines. Finally, and unlike EMF which is not thread-safe, KMF provides a built-in support for in-memory safe concurrent access to models.

[17] http://www.eclipse.org/cdo/

hal-00714558, version 1 - 5 Jul 2012

KMF is still at an early stage of existence and needs to be improved through usage. Future work on KMF already addresses the limitations and points discussed in section 5. Independently from the improvement of existing features of KMF, we think that additional tools could promote its adoption.

Set operations. Model merging or model comparison are common operations implemented by tools that use models as representations of their internal data. Implementing mergers or comparators is often a complex, lengthy and error prone task. As a future work, we plan to offer the possibility to generate metamodel specific set operations such as union, difference or intersection. These operations could decrease the complexity of implementing model mergers.

Customizable generation plugin. In its current implementation, the plugin allows for generation of all features (model, cloner, loader and serializer) or only model and cloner. This customization of the plugin behavior will be improved to enable the separate generation of each feature. Moreover, the loader and serializer generators are hard coded in the plugin. It is thus problematic to use other generators to create loaders and serializers that use another serialization format (namely XMI). In the future, we plan to improve the plugin parameterization to allow users to change the generators. This will also enable the seamless integration of other generators (e.g. to create set operations).

References

1. Cyril Ballagny, Nabil Hameurlain, and Franck Barbier. Mocas: A state-based component model for self-adaptation. In *Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009, San Francisco, California, USA, September 14-18, 2009*, pages 206–215. IEEE Computer Society, 2009.
2. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.
3. Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. 5525:1–26, 2009.
4. Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.
5. François Fouquet, Olivier Barais, and Jean-Marc Jézéquel. Building a kermeta compiler using scala: an experience report. In *Workshop Scala Days 2010*, Lausanne, Switzerland, 2010.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
7. John C. Georgas, André van der Hoek, and Richard N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10):52–60, October 2009.

hal-00714558, version 1 - 5 Jul 2012

8. Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In Richard Paige, Alan Hartman, and Arend Rensink, editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer Berlin / Heidelberg, 2009.
9. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
10. Bernhard Merkle. Textual modeling tools: overview and comparison of language workbenches. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 139–148, New York, NY, USA, 2010. ACM.
11. Brice Morin, Olivier Barais, Grégory Nain, and Jean-Marc Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.
12. P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *MoDELS'05: 8th Int. Conf. on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica, Oct 2005. Springer.
13. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.
14. Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.
15. Fredrik Seehusen and Ketil Stlen. An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations*, volume 6707 of *Lecture Notes in Computer Science*, pages 152–166. Springer Berlin / Heidelberg, 2011.
16. Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

Author manuscript, published in "DAIS 2012 (2012)"

hal-00688707, version 1 - 18 Apr 2012

Dissemination of reconfiguration policies on mesh networks

François Fouquet, Erwan Daubert, Noël Plouzeau,
Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel
University of Rennes 1, IRISA, INRIA Centre Rennes
Campus de Beaulieu, 35042 Rennes, France
{Firstname.Name}@inria.fr

Abstract. Component-based platforms are widely used to develop and deploy distributed pervasive system that exhibit a high degree of dynamicity, concurrency, distribution, heterogeneity, and volatility. This paper deals with the problem of ensuring safe yet efficient dynamic adaptation in a distributed and volatile environment. Most current platforms provide capabilities for dynamic local adaptation to adapt these systems to their evolving execution context, but are still limited in their ability to handle distributed adaptations. Thus, a remaining challenge is to safely propagate reconfiguration policies of component-based systems to ensure consistency of the architecture configuration models over a dynamic and distributed system. In this paper we implement a specific algorithm relying on the *models at runtime* paradigm to manage platform independent models of the current system architecture and its deployed configuration, and to propagate reconfiguration policies. We evaluate a combination of gossip-based algorithms and vector clock techniques that are able to propagate these policies safely in order to preserve consistency of architecture configuration models among all computation nodes of the system. This evaluation is done with a test-bed system running on a large size grid network.

1 Introduction

Nowadays, the increasing use of Internet of Things devices for computer supported cooperative work leads to large systems. As these devices use multiple mobile networks, these systems must deal with concurrency, distribution, and volatility issues. This volatility requires dynamic auto-adaptation of the system architecture, in order to provide domain specific services continuously. Tactical information and decision support systems for on field emergency management are perfect examples of such highly dynamic systems. Indeed, these multi-user interactive systems built on mobile devices need frequent changes of architecture to deal with rapid system evolution (*e.g.* scale up or scale down of team, download of new software modules by the device user) or to cope with network disconnections. For such systems, the traditional design process "design, code, compile, test, deploy, use, iterate" does not work.

Dynamic adaptation, pursuing IBM's vision of autonomic computing, is a very active area since the late 1990's - early 2000's [9]. Modern component-based systems [15,4] provide a reflection and intercession layer to dynamically reconfigure a running system. But the reconfiguration process remains complex, unreliable and often irreversible in a volatile and distributed context. The use

hal-00688707, version 1 - 18 Apr 2012

of model-driven techniques for managing such run-time behavior (named models@runtime [3]) helps to handle software reconfiguration. Models@runtime basically pushes the idea of reflection [14] one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance to trigger reconfigurations. Kevoree is our open-source dynamic component model[1], which relies on models at runtime to properly support the dynamic reconfiguration of distributed systems. The model used at runtime reflects the global running architecture and the distributed topology. In Kevoree, when a distributed node receives a model update that reflects the target running architecture, the node extracts the reconfigurations that affect it and transform them into a set of platform reconfiguration primitives. Finally, it executes them and propagates the reflection model to other nodes as a new consistent architecture model.

In a highly distributed and volatile environment, one of the challenges is the propagation of reconfiguration policies. Handling concurrent updates of shared data is a second challenge to be solved, as two nodes can trigger concurrent reconfigurations. Consistent dissemination of models at runtime in distributed systems requires a synchronization layer that solves these two challenges: information dissemination and concurrent update. Research in the field of peer-to-peer communication has produced many algorithms to deal with information dissemination in a volatile context [6]. Many paradigms are available to deal with this concurrent data exchange problems (e.g. vector clocks [7]). In this paper, we adapt a combination of gossip-based algorithms and vector clocks techniques to safely propagate reconfiguration policies by preserving architecture models consistency between all computation nodes of a distributed system. We have implemented a specific algorithm, which propagates configuration changes in a consistent manner in spite of frequent node link failures, relying on its payload of configuration data to improve its efficiency. We provide qualitative and quantitative evaluations of this algorithm, to help answering the following questions: (i) What is the influence of communication strategy on the propagation delay of models? (ii) Does a high rate of node link failure prevent the propagation of models and what is the impact of link failures on propagation delays? (iii) Does the algorithm detect concurrent updates of models and does it handle reconciliation correctly?

The remainder of this paper is organized as follows. Section 2 presents the background of this work. Section 3 details the combination of a gossip-based algorithm and the vector clock techniques used to preserve architecture models consistency between all computation nodes of the system. Section 4 details our experiments to evaluate this combination. Section 5 discusses articles, ideas and experimental results related to our work. Finally, Section 6 concludes this paper and presents ongoing work.

2 Background

Kevoree is an open-source dynamic component model[1], which relies on models at runtime [3] to properly support the dynamic adaptation of distributed sys-

[1] `http://kevoree.org`

hal-00688707, version 1 - 18 Apr 2012

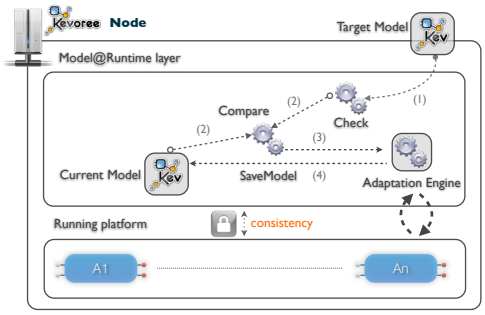


Fig. 1. Models@Runtime overview

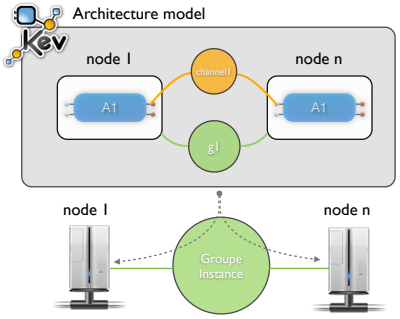


Fig. 2. Distributed reconfigurations

tems. Figure 1 presents a general overview of models@runtime. When changes appear as a new model (a target model) to apply on the system, it is checked and validated to ensure a well-formed system configuration. Then it will be compared with the current model that represents the running system. This comparison generates an adaptation model that contains the set of abstract primitives to go from the current model to the target one. Finally, the adaptation engine executes configuration actions to apply these abstract primitives. If an action fails, the adaptation engine rollbacks the configuration to ensure system consistency. Kevoree has been influenced by previous work that we carried out in the DiVA project [14]. With Kevoree we push our vision of models@runtime [14] farther. In particular, Kevoree supports distributed models@runtime properly. To this aim we introduce the *Node* concept in the model to represent the infrastructure topology. Kevoree includes a *Channel* concept to allow for multiple communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node) obey the Type Object pattern [8] to separate deployment artifacts from running artifacts. Kevoree supports multiple kinds of execution node technology (e.g. Java, Android, MiniCloud, FreeBSD, Arduino, ...).

Kevoree also introduces a dedicated concept named *Group*, to encapsulate platform synchronization algorithms. *Group* allows to define communication channels between nodes to propagate reconfiguration policies (i.e. new target model). This *Group* concept also encapsulates a dedicated protocol to ensure specific synchronization policies (e.g. Paxos derived algorithms for total order synchronization, Gossip derived algorithms for partial order and opportunistic synchronization). *Group*s can be bound to several nodes (named members), allowing them to explicitly define different synchronization strategies for the overall distributed system. This architecture organization is illustrated in Figure 2. In addition, a *Group* also defines a scope of synchronization, *i.e.* it defines which elements of the global model must be synchronized for the group's members. This avoids to globally share model@runtime models.

P2P algorithm and mesh network Schollmeier [16] defines a peer-to-peer network as "a distributed network architecture, where participants of the network share a part of their resources, which are accessible by the other peers directly, without passing intermediary entities". He also provides the following

hal-00688707, version 1 - 18 Apr 2012

distinction: *hybrid* peer-to-peer networks use a central entity, while *pure* peer-to-peer networks have no such entity. According to Wikipedia, a mesh network is "a type of network where each node must not only capture and disseminate its own data, but also serve as a relay for other nodes, that is, it must collaborate to propagate the data in the network". In these network topologies, gossip-like algorithms are good solutions to disseminate data.

Concurency data management for distributed message passing applications Distributed systems consist of a set of processes that cooperate to achieve a common goal. Processes communicate with data exchanges over the network, with no shared global memory. This leads to well known and difficult problems of causality and ordering of data exchanges. Solutions are known to cope with this problem: Lamport [10] defines an event order using logical clocks by adding a logical time on each message sent. Another solution was coinvented by Fidge [7] and Mattern [13], using a vector of logical clocks. In many cases the vector clock technique is the most appropriate solution to manage a partial order and concurrency between events[2], e.g. in distributed hash table systems such as Voldemort[2]).

Synthesis In our vision of distributed environments, system management is decentralized, allowing each peer to build, maintain and alter the overall architecture and platform models at runtime. Because of nodes volatility, ensuring consistency during reconfiguration is a critical task. We use Kevoree and the notion of *Group* to encapsulate platform synchronization algorithms with gossip and vector clock techniques.

3 An algorithm to disseminate reconfiguration policies

Each node holds a copy of the model that describes the overall system configuration. This system model contains a description of the nodes that currently compose the system, of components that are installed on each node and of network links between nodes. It also contains all information about *groups*. A *group* is the unit of model consistency for the models at runtime technique. Each node involved in model consistency includes several named *group instances*, which participates in the distributed model management for the local node. Part 1 of the algorithm provides the data definition for one node.

In addition to the information given by the model, each group instance maintains specific information (see algorithm's Part 2): a group id, a local copy of the model and the local node id. It also stores its current vector clock, a score for each of its neighbors and a boolean attribute to record whether the model has changed since the last time another node requested the local node's vector clock. The score of the neighbors is used to select the more interesting one when the local node looks for new reconfigurations.

[2] `http://project-voldemort.com`

hal-00688707, version 1 - 18 Apr 2012

Algorithm Part 1 DEFINITIONS

Message ASK_VECTORCLOCK, ASK_MODEL, NOTIFICATION
Type VectorClockEntry := <id: String, version ∈ ℕ>
Type Node // represents a node on the system
Type Model // represents a configuration of the system
Set Group:= {node: Node}
Set IDS(g: Group):= {id: String | ∃ node: Node, node ∈ g & node.name = id}
Set Neighbors(originator: Node, g:Group):= {node: Node | node ∈ g & originator ∈ g}
Set VectorClock(originator: Node, g: Group):= {entry: VectorClockEntry | entry.id == originator.name}
⋃ {entry1: VectorClockEntry |∃ node: Node, node != originator & entry1.id ∈ IDS(g) & node ∈ g}
Set VectorClocks(originator: Node, g: Group):= {vectorClock: VectorClock(originator, g)}

Main algorithm (see algorithm's Part 3). When a change appears on the model stored in a node, the corresponding group instance is notified. The group instance then sends notification to all its neighbors. These neighbors in turn may send a message to the current node, to ask for model update information. As the underlying communication network is volatile and unreliable, some notifications can be lost and not received by some members of a group. To deal with these losses, each member of a group asks periodically a chosen group member for changes. Since a model is a rather large data, group instances ask for the vector clock of the remote instance first, in order to decide if a model transfer is needed. More precisely, after comparing the vector clock received with its own vector clock, a group instance will request a model if both vector clocks are concurrent or if the vector clock received is more recent than its local one. Here concurrency means that each local and remote model have different changes which dot not appear on the other. A vector clock is more recent than another if some changes appear on it but not on the other. Upon reception of a model, the group instance compares the model's vector clock and the local clock again. If the local vector clock is older, the local node updates its local clock and also updates the local copy of the model using the model just received. If the vector clocks are concurrent then the group must resolve this concurrency at the model level to compute the correct model and then update the vector clock accordingly.

Functions SelectPeer (see Algorithm Part 4) In addition to this mechanism, each node pulls periodically one of its neighbors, in order to cope for lost notifications. The selection of the neighbor to pull is controlled by a score mechanism: a score is assigned to each peer by the group instance and the selection of the peer is done according to the smaller score. The score of the node grows when it is selected or when the network link to access this node seems to be down. The

Algorithm Part 2 STATE

g: Group ; changed: Boolean
currentModel: Model // local version of system configuration
localNode: Node // representation of local node
currentVectorClock ∈ VectorClocks(localNode, g)
scores := {<node: Node, score>, node ∈ Neighbors(localNode, g) && score ∈ ℕ}
nbFailure := {<node: Node, nbFail>, node ∈ Neighbors(localNode, g) && nbFail ∈ ℕ}

hal-00688707, version 1 - 18 Apr 2012

Algorithm Part 3 ALGORITHM

```
On init():
  vectorClock ← (localNode.name, 1)
  scores ← {Neighbors(localNode, g) × {0}}
  changed ← false
On change (currentModel):
  ∀ n, n ∈ Neighbors(localNode, g) → send (n, NOTIFICATION)
  changed ← true
Periodically do():
  node ← selectPeerUsingScore()
  send (node,ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode, g), NOTIFICATION):
  send (neighbor, ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode,g), remoteVectorClock ∈ VectorClocks(neightbor, g)):

  result ← compareWithLocalVectorClock (remoteVectorClock)
  if result == BEFORE || result == CONCURRENTLY then
     send (neighbor, ASK_MODEL)
  end if
On receive (neightbor ∈ Neighbors(localNode,g),vectorClock ∈ Vectorclocks(neighbor, g), model)):

  result ← compareWithLocalVectorClock (targetVectorClock)
  if result == BEFORE then
     updateModel(model)
     mergeWithLocalVectorClock(vectorClock)
  else if result == CONCURRENTLY then
     resolveConcurrently(vectorClock, model)
  end if
On receive (neighbor ∈ Neighbors(localNode,g), request):
  if request == ASK_VECTORCLOCK then
     checkOrIncrementVectorClock()
     send (neighbor, currentVectorClock)
  end if
  if request == ASK_MODEL then
     checkOrIncrementVectorClock()
     send (neighbor, <currentVectorClock,currentmodel>)
  end if
```

down link detection relies on a synchronization layer. This layer uses model information to check for all available peers periodically and then to notify the group instance of unreachable nodes. A peer score takes into account the duration of unavailability of the peer. When the peer becomes available, this number is reset to 0: restored availability clears the failure record. Indeed, as the system uses a sporadic and volatile network, peers often appear and disappear and most of the time disappearance events are not causally connected.

Functions about vector clocks (see Algorithm Part 5) Our algorithm relies on vector clocks to detect changes in remote configuration models. When a local update of the model appears, a boolean called *changed* is set to true to ensure that upon a vector clock request from another node the group instance will increment by one its version id in its local vector clock before sending it to the requesting peer. In case of concurrent updates of models we rely on the use of the reflexivity provided by the model at runtime to solve the conflict. Priority is given to information about the nodes already reached and affected by the update. Any node detecting a conflict will merge these models and their associated vector clocks to store it as its current state. A reasoning upper layer

hal-00688707, version 1 - 18 Apr 2012

Algorithm Part 4 SelectPeer

```
Function selectPeerUsingScore()
  minScore := ∞ ; potentialPeers := {}
  for node → Neighbor(localNode, g) do
    if node ! = localNode && getScore(node) < minScore then
      minScore := getScore(node)
    end if
  end for
  for node → Neighbor(localNode, g) do
    if node ! = localNode && getScore(node) == minScore then
      potentialPeers := potentialPeers ⋃ {node}
    end if
  end for
  node := select randomly a node from potentialPeers
  updateScore(node)
  return node
Function getScore(node ∈ Neighbors(localNode, g))
  return scores(node)
Function updateScore(node ∈ Neighbors(localNode, g))
  oldScore := getScore(node)
  scores := scores ⋃ {node, oldScore + 2 * (nbFailure + 1)} \ {node, oldScore}
```

will then compute an update from this merged model by reading the model and correcting it. Description of this reasoning layer is beyond the scope of this paper and vector clocks merge and comparison is already defined on previous works on vector clocks [7] and [13].

Algorithm Part 5 FUNCTIONS

```
Function checkOrIncrementVectorClock()
  if changed == true then
    ∀ entry, entry ∈ currentVectorClock & entry.id == localNode.name ⇒ entry.v ← entry.v + 1
    changed ← false
  end if
Function compareWithLocalVectorClock(targetVectorClock∈VectorClocks(n ∈ neighbors(localNode, g), g)) // for details, please look at http://goo.gl/0tdEc
Function mergeWithLocalVectorClock(targetVectorClock ∈ VectorClocks(n ∈ neighbors(localNode, g), g)) // for details, please look at http://goo.gl/axbJN
Function resolveConcurrency(targetVectorClock ∈ VectorClocks(n ∈ neighbors(localNode, g), g)) // for details, please look at http://goo.gl/bFTeH
Function updateModel(model ∈ Models)
  currentModel ← model
```

4 Evaluation

We have performed qualitative and quantitative evaluations of our algorithm, aiming at measuring the following indicators: (1) model propagation delay; (2) resilience to node link failure; (3) ability to detect concurrent models and to handle reconciliation. For each indicator we have set up an experimental protocol, using the firefighter tactical information case study metrics to simulate the system behaviour on a grid in different configurations. Although target platforms will be pervasive embedded systems, we have chosen a large scale grid as an evaluation testbed. The use of a grid allows us to stress the algorithm by setting up a large number of nodes but it also brings us more control over the parameters of the experiment e.g. network failure simulations. In this way experiments are reproducible, and reproducibility is essential to our experimental protocol. On-field validation is an ongoing work.

hal-00688707, version 1 - 18 Apr 2012

4.1 Common experimental protocol

Validation experiments share a common experimental protocol. Each experiment uses a set of logical Kevoree node deployed on physical nodes within a computer grid. Each Kevoree logical node is instantiated in a separate Java Virtual Machine and use the reference Kevoree implementation for JavaSE. The experimental grid is an heterogeneous grid that contains nodes of mixed computational power and type. Each node is connected to a local area network at 100 MB/s.

Topology model All our experiments take a bootstrap model as input, which describes the current abstract architecture (*i.e.* in its platform independent form). This abstract model contains information on node representations, node logical links and node communication group instances and relationships. This node set and these relationships describe a topology of the system, which is used by our synchronization algorithm. In order to improve the simulation of a firefighter tactical information case study, we use a random generator to create topology models that are organized in a cluster of clusters. In this way it is easier to simulate non-direct communication (*i.e.* node A cannot communicate directly with node B but must pass through node C).

Global time axis traces In order to track the propagation of new configurations in this distributed system, we decorate the algorithm with a logger. This logger sends a trace for each internal state change (*i.e.* new configuration or configuration reconciliation). These traces describe the current state of the group, namely the new vector clock, the identification of the peer originator of change and the network metrics used. In order to exploit temporal data on these traces without ensuring a global grid time synchronization we use a logger with a global time axis based on Java Greg Logger[3]. More precisely, this type of logger is based on a client server architecture. The server clock manages the global time reference. All clients periodically synchronize with the server, allowing it to store client latencies by taking into account clocks shift and network time transfer observed. Traces are emitted asynchronously by the client to the server, which then makes time reconciliation by adding the last value of latency observed for this client. All traces emitted by the server are therefore all time stamped accurately with the clock of the server. Finally, traces are chained by an algorithm to meet the following heuristic: a trace follows another one if it is the first occurrence that contains in its vector clock the originator node with its precise version number. Thus the final result for each experiment is a linked trace list on which we can precisely compute temporal results.

Communication modes We reuse mainly two classical exchange patterns to build our algorithm.
Pooling period term is associated with the time elapsed between two active synchronizations, and is initiated by a group member to another. In this synchronization step a vector clock and/or a model is sent back to the initiator.

[3] `http://code.google.com/p/greg/`

hal-00688707, version 1 - 18 Apr 2012

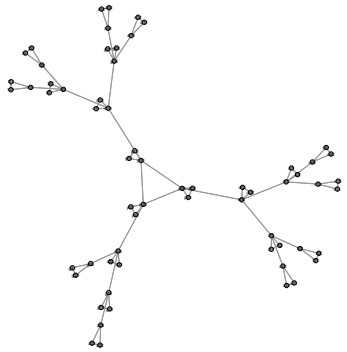


Fig. 3. Topology model of exp 1

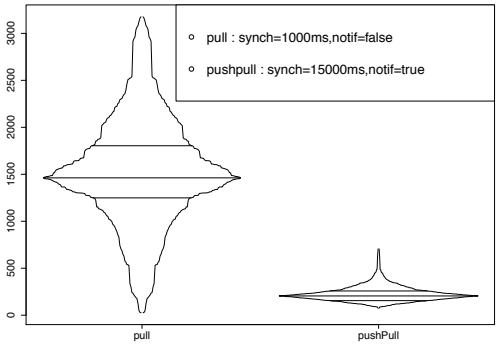


Fig. 4. Delay/hop(ms)

The Push/Pull technique is an association of the pooling active synchronization and an event-driven notification mechanism. This operation adds to the pooling mode a sending step to every reachable group member.

4.2 Experimental studies

Propagation delay versus network usage This first experiment aims at performing precise measurements of the capacity to disseminate model configurations. These measures will take care of the propagation delay and the network usage properties.

Experimental protocol As described in the common protocol subsection, measurements are performed on a computer grid. The probes injected in the Java implementation collect propagation delay and network occupation. After a bootstrap step on a topology model, a node chosen at random reconfigures its local model with a simple modification. In practice this reconfiguration step computes a new model, moving a component instance from one node to another chosen randomly. This new model is stored in the node, and the reconfiguration awaits propagation by the algorithm. This new configuration is tagged with the identification of reconfiguration originator. Figure 3 shows the topology model used for multi hop communication in the 66 nodes of this configuration. In this experiment, the network topology is static. No node joins or leaves the system.

The experiment is driven by the following parameters:(1) delay before starting an active check of the peers update (model synchronization);(2) activation of sending of change notification messages. To evaluate the impact of the second parameter, the experiment is run twice. In the first run, notifications are not used and the active synchronization delay is set to 1000 ms. In the second run, notifications are used and active synchronization delay is 15 s. In both cases, a reconfiguration is triggered every 20 seconds and each reconfiguration run takes 240 seconds, resulting in 12 reconfiguration steps.

Analysis The observed per hop propagations delays are presented as a percentile distribution (see Graph 4). The values displayed are the raw values of absolute

hal-00688707, version 1 - 18 Apr 2012

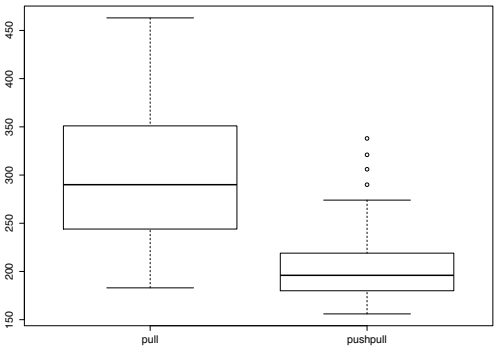


Fig. 5. Network usage/node(in kbytes)

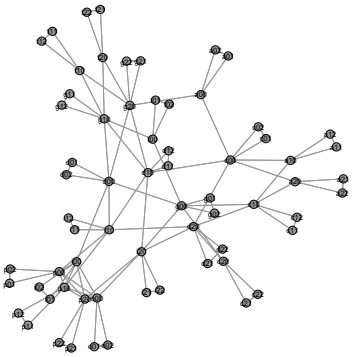


Fig. 6. Topology of exp 2

time logged divided by the minimum number of hops between the target and originator of the reconfiguration (the minimum being computed using a Bellman-Ford algorithm [5]). The traffic volume from protocol messages is shown in Figure 5 in KB per node per reconfiguration; the volume does not include payload. Absolute values of network consumption depends highly of implementation. Results presented here are from the Java version and can be vastly improved when targeting embedded devices like microcontrollers.

The use of notification reduces the propagation delay significantly: the average value decreases from 1510 ms/hop to 215 ms/hop. In addition, percentile distribution shows that the standard deviations of propagations are lower with in the version with notification. Thus this version of the algorithm has better scalability for large graph diameters.

However, in comparing the push pull and the push algorithm, the use of notification on network usage is not as significant. Analysis shows that these results are affected by cycles in the topology. When using notification of change, nodes in cycles will create parallel branches configuration diffusion. This in turn will increase the number of conflict resolution to be done, and these resolutions increase network consumption unnecessarily, by exchanging the same model version. When notifications are not used, pooling delays are large enough to avoid this concurrent configuration "flood". As the payload is a model with topology information, the notification algorithm could use this information to prevent flood. This solution will be studied in future work.

Failures impact on propagation delay A mobile mesh network such as the one used in a firefighter tactical information system is characterized by a large number of nodes that are often unreachable. We designed our algorithm to cope with these network problems. The second experiment described below tests the ability of the algorithm to disseminate new models in a mesh network with different failure rates.

Experiment protocol The experiment protocol is similar to the first experiment's one. The topology model is enhanced to provide a mesh network with many

hal-00688707, version 1 - 18 Apr 2012

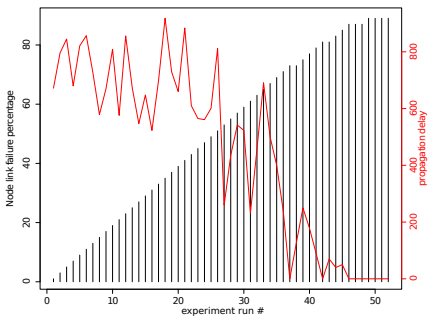


Fig. 7. Failure results

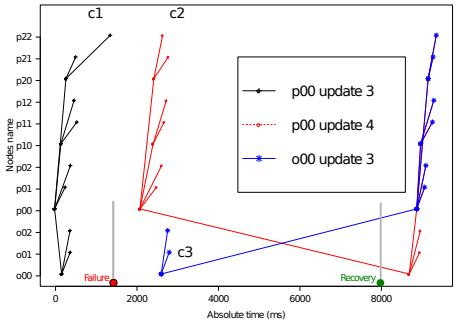


Fig. 8. Concurrent update

different routes between nodes (see Figure 6). At each run a modified model is pushed on a random node. The reconfiguration is similar to the previous experiment. During each run, additional failures are simulated on links between two nodes, according to a Poisson distribution. The failure rate is increased at each run, thus the number of reachable nodes decreases. To perform this failure simulation we inject probes, which also monitor synchronization events. At each run, the list of theoretically reachable nodes is computed and the initiator node waits for synchronization events from theses nodes. When all events have been received we compute the average propagation delay. In short, this experiment aims at checking that every theoretically reachable node receives the new configuration.

Analysis Figure 7 shows results of experiment #2. The histogram shows the rate of network failure for each run. The red curve displays the average propagation delay to reachable nodes (in milliseconds). Above a network failure of 85% the node originator of the reconfiguration is isolated from the network and therefore we stop the execution. With a failure rate under 85% every node receives the new configuration and we can compute the propagation delay.

Concurrency reconfiguration reconciliation Our third experiment addresses the problem of reconciliation and conflict detection between concurrent model updates. This problem occurs often in the firefighter tactical information case study architecture because of the sporadic communication capabilities of our network of nodes. As a node can stay isolated for some time, reconfiguration data no longer reaches it. Furthermore, local reconfigurations can also occur in its subnetwork. Connection restoration may produce conflicting concurrent model updates. We rely on vector clocks to detect these conflicts and on the conflicting model updates themselves. Experiment #3 aims at checking the behaviour of our algorithm in this conflicting updates situation.

Experiment protocol The experiment protocol is based on experiment #2. We use a similar grid architecture but with only 12 nodes. An initial reconfiguration (c1)

hal-00688707, version 1 - 18 Apr 2012

is launched on the *p00* node just after the bootstrap phase. All network links are up. Then a fault is simulated on the link between nodes *p00* and *o00*. Nodes *o00*, *o01*, *o02* are then isolated. A new model is then pushed on node *p00* (c2) and a different one on node *o00* (c3). A delay of 1000 ms separates each reconfiguration and the algorithm is configured with a notification and a pooling period of 2000 ms.

Analysis Figure 8 shows results of experiment #3, which are derived from our branching algorithm traces. Three reconfigurations are represented as a succession of segments that show the propagation of updates. The first reconfiguration on the healthy network is represented in black (at time 0). Reconfiguration pushed on *o00* (at time 2500) is represented in blue and the second reconfiguration pushed on *p00* (time 2000) in red. The first reconfiguration propagates seamlessly to all nodes. At time 1500 a network failure is simulated. The second model given to *p00* is propagated to all nodes except nodes reachable through *o00* only. Similarly, the second model pushed on node *o00* is not propagated to nodes after *p00*. At time 8000 we cancel the network failure simulated at time 1500. After a synchronization delay (380ms) we observe the branching of the two concurrent models as well as propagation of the merged version (purple line).

5 Discussion and related work

Our approach is dedicated to model at runtime synchronization, and combines commonly used paradigms in distributed computing like vector clocks (e.g. used in distributed hash table frameworks) and gossiping (e.g. used in social network graph dissemination). This section discusses our experimental results and compare them to other related work.

Vector clock size. Our first experiment measures the size of data exchanged during the reconfiguration step, as well as the time required to perform this reconfiguration. Figure 5 shows that the model@runtime synchronization overhead is significant, and this is mostly due to vector clock size. Many studies aim at reducing the data size of vector clocks, especially when synchronizing an unbounded number of peers. Sergio and al [1] proposed the Interval Tree Clocks to optimize the mapping between the node identifier and its version. Our algorithm takes advantage of the model payload to allocate dynamic identifiers to nodes. Data such as node names or network identifications are stored in the payload itself and with this information we can already improve vector clocks. However, we plan to implement the interval tree clocks' fork and join model in the future. The size of exchanged data depends on the number of nodes and therefore modularization techniques are needed to maintain scalability and manage large mesh networks. Our approach addresses this need by exploiting the group structure of Kevoree. Each group instance synchronizes with a subset of nodes only, to keep the size of the vector clock under control.

Distributed reconfiguration capability. Concurrency management is a key problem in distributed systems. Many peer to peer systems solve it by having a single point of update for a given piece of data, limiting concurrent access to a one writer/many readers situation for that data. Realistic distributed configuration management is a many writers/many readers situation, because reconfigurations often involve more than one node. The simplest solution to this problem would use a single point for new configuration computation and dissemination start. As it avoids concurrency, such a system has a central point

hal-00688707, version 1 - 18 Apr 2012

of failure incompatible with our use case. More advanced approaches such as the one proposed in [17] use distributed coordination techniques such as consensus to build the new configuration. They proposed an approach that allows the distributed nodes to collaborate to build the new configuration. Each node is responsible for building its local configuration. Configuration propagation is then done using a gossip algorithm without the need of concurrency management, since new configurations can be disseminated from a single originator node only. This approach based on a single source is unusable in our use case, because the sporadic nature of the nodes prevent their participation in a global consensus. On the contrary, our technique presented in this paper lets the distributed configuration evolve freely, even for nodes are isolated in unreachable groups. Every node can then compute a new global model that can be issued concurrently. Some approaches in distributed hash table implementations also rely on fully distributed data dissemination, e.g. Voldemort, where table modifications can occur in several nodes. This allows for service operation in degraded mode in the case of node disconnections. However, concurrency management must be managed separately. GossipKit [12] proposes a generic framework to evaluate and simulate gossip-derived algorithms. The project contains a minimal extensible event-based runtime with a set of components to define dedicated gossip protocols. We plan to integrate the GossipKit API in order to evaluate our algorithm on a GossipKit simulator.

Inverted communication and propagation delay. In our approach we reverse the traditional communication strategy of a gossip algorithm (*push* approach). New configurations are not directly pushed to the neighbours but they are stored instead, waiting for an active synchronisation by the neighbour (*pull* approach). This strategy lessens the impact of down network links on propagation delay, as shown by our experiment results on Figure 7. In addition, this enables message replay because a configuration is stored until neighbor connectivity is reestablished. These two properties are particularly useful for unreliable mesh networks. However, pull approaches have higher propagation time, but when combined with an observer pattern (a lazy push/pull approach) our results show that the gains are significant while keeping the interesting properties of pull. This experimental result is consistent with Leitao *et al* [11], which details several communication strategy for gossip algorithms.

6 Conclusion

In this paper we proposed a peer to peer and distributed dissemination algorithm to manage dynamic architectures based on the *models at runtime* paradigm. This algorithm is part of a larger framework that manages the continuous adaptation of pervasive systems. Using experimental results we have shown how our approach enhances reliability and guarantee of information delivery, by mixing and specializing different distributed algorithms. Our propagation algorithm relies on its payload (a model of the system) to overcome limits of vector clocks and to handle peer to peer concurrency conflicts. Thanks to the protocol layer based on vector clocks, a system architecture model propagated by the algorithm is always consistent, even on complex mesh network topologies. When concurrent updates are detected, the model at runtime layer is able to reconcile these updates to provide a valid architecture. By allowing each node to compute a new

hal-00688707, version 1 - 18 Apr 2012

configuration, our approach supports dynamic adaptation on peer to peer networks without any central point of failure. This experimental demonstration of resilience on sporadic networks allows integration of our approach into adaptive architectures such as a firefighters tactical information system. In this direction, we are currently designing a dynamically scalable tactical information system in collaboration with a department of firefighters of Brittany; this system is a multi-user, real time decision system for incident management.[4].

Acknowledgment The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

1. P.S. Almeida, C. Baquero, and V. Fonte. Interval tree clocks: A logical clock for dynamic systems. *Principles of Distributed Systems*, page 259274.
2. R. Baldoni, M. Raynal, and U..R.L.S. DIS. Fundamentals of distributed computing. *IEEE Distributed Systems Online*, 3(2):1–18, 2002.
3. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@runtime. *IEEE Computer*, 42(10):22–27, 2009.
4. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
5. C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *SIGCOMM Comput. Commun. Rev.*, 19:224–236, August 1989.
6. P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE computer*, 37(5):60–67, 2004.
7. C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th ACSC*, volume 10, pages 56–66, 1988.
8. Ralph Johnson and Bobby Woolf. The Type Object Pattern, 1997.
9. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
11. J. Leitão, J. Pereira, and L. Rodrigues. Gossip-based broadcast. *Handbook of Peer-to-Peer Networking*, pages 831–860, 2010.
12. S. Lin, F. Taïani, and G. S. Blair. Facilitating gossip programming with the gossipkit framework. In *DAIS*, 2008.
13. F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.
14. B. Morin, O. Barais, J-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.
15. G.S. Raj, PG Binod, K. Babo, and R. Palkovic. Implementing service-oriented architecture (soa) with the java ee 5 sdk. *Sun Microsystems, revision*, 3, 2006.
16. R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102. IEEE, 2001.
17. D. Sykes, J. Magee, and J. Kramer. Flashmob: distributed adaptive self-assembly. In *Proceeding of the 6th SEAMS*, pages 100–109. ACM, 2011.

[4] More details on this project can be found in `http://kevoree.org/related_projects`

Author manuscript, published in "15th International ACM SIGSOFT Symposium on Component Based Software Engineering (2012)"

A Dynamic Component Model for Cyber Physical Systems

Francois Fouquet, Olivier Barais,
Noel Plouzeau, Jean-Marc Jezequel
IRISA, University of Rennes1, France
firstname.name@irisa.fr

Brice Morin Franck Fleurey
SINTEF ICT, Oslo, Norway
firstname.name@sintef.no

ABSTRACT

Cyber Physical Systems (CPS) offer new ways for people to interact with computing systems: every thing now integrates computing power that can be leveraged to provide safety, assistance, guidance or simply comfort to users. CPS are long living and pervasive systems that intensively rely on microcontrollers and low power CPUs, integrated into buildings (e.g. automation to improve comfort and energy optimization) or cars (e.g. advanced safety features involving car-to-car communication to avoid collisions). CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. This paper presents μ-Kevoree, the projection of Kevoree (a component model based on models@runtime) to microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. Its evaluation regarding key criteria in the embedded domain (memory usage, reliability and performance) shows that, despite a contained overhead, μ-Kevoree provides the advantages of a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) compared to traditional techniques for dynamic firmware upgrades.

Categories and Subject Descriptors

D.2 [**Software Engineering**]; D.2.8 [**Software Engineering**]: Software Architectures—*Domain-specific architectures; Languages*

Keywords

Component-based software engineering ; Autonomic computing ; Embedded software ; Software architecture

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CBSE'12, June 26–28, 2012, Bertinoro, Italy.
Copyright 2012 ACM 978-1-4503-1345-2/12/06 ...$10.00.

1. INTRODUCTION

Over the last decades, the Internet has undergone dramatic changes, moving from a rather static Internet of Content, to an always more complex, dynamic and ubiquitous mix of Internet of People, Services (IoS), and Things (IoT) [1]. Based on this infrastructure, which ranges from large datacenters and cloud servers to an heterogeneous and ever growing set of things (smartphones, sensors, etc) operated by resource-constrained CPUs and microcontrollers, Cyber Physical Systems (CPS) have emerged. CPS are long living and pervasive systems that rely intensively on microcontrollers and low power CPUs, integrated into buildings and cities (automation to improve comfort, safety and energy optimization), cars (advanced safety features involving car-to-car communication to avoid collisions), and so on.

CPS operate in volatile environments where nodes should cooperate in opportunistic ways and dynamically adapt to their context. In a car-2-car scenario, 2 cars (or more) approaching the same intersection should be able to synchronize in a reasonably short delay to share information about their own context and configuration, then take distributed decisions *e.g.* on the precedence order to cross the intersection. In a building automation scenario, users working or living in the building should be able to customize their working or living environment according to their desires and needs, *e.g.* using their smartphones or tablets to adjust the intensity of the lights according to the ambient light, etc. In a factory chain scenario, robots operating on the chain should be able to adapt and cope with failures, instead of shutting down all the chain when a failure is detected. Depending on the context, it is necessary to dynamically adapt both the software and the way the CPS are configured, as things are containers that can host services.

Dynamic adaptation, pursuing IBM's vision of autonomic computing, is a very active area since the late 1990's - early 2000's [20]. However, many existing techniques concentrate on the adaptation of rather powerful nodes, which are typically able to run a Java Virtual Machine. Adaptation of resource-constrained devices such as microcontrollers has received less attention. These resource constraints prevent the use of standard operating systems, middlewares and frameworks, making the design of adaptive software for microcontroller a challenging task. In practice, microcontroller code is most of the time developed by using low-level programming languages and by following ad-hoc manual trial and error processes; these processes includes extensive testing of the resulting software in its target environment. While this might be acceptable to build static, dedicated applications,

hal-00713769, version 1 - 2 Jul 2012

hal-00713769, version 1 - 2 Jul 2012

this is not a practical solution for CPS, because CPS operate in an open and dynamic environment, where the target environment cannot be foreseen at design-time.

Kevoree leverages and extends state-of-the-art approaches to scale CBSE principles horizontally (distribution between a large set of nodes) and vertically (from cloud-based nodes to microcontrollers). This paper focuses on μ-Kevoree, a mapping of Kevoree concepts for microcontrollers. μ-Kevoree pushes dynamicity and elasticity concerns directly into resource-constrained devices. In particular, this paper details the challenges of mapping such a large component model onto microcontroller-based architectures. We explain the tradeoffs that were used to obtain a useful solution coping with stringent resource constraints. This dynamic component model for resource-constrained systems has been thoroughly benchmarked against key criteria that are specific to the embedded software domain (memory usage, reliability and performance). Our model has also been applied to a real-life case study. The evaluation of μ-Kevoree for these key criteria show that, despite a contained overhead, μ-Kevoree provides a dynamically reconfigurable component-based model (safe, fine-grained, and efficient reconfiguration) with a limited overhead with respect to static approaches.

This paper is organized as follows. Section 2 presents the Kevoree component model and Section 3 details the challenges of mapping dynamic component model concepts to resource-constrained microcontrollers. Section 4 then explains how we ported Kevoree to these challenging platforms, and details the necessary tradeoffs. This new version of Kevoree is validated in Section 5 through a set of atomic benchmarks. Section 6 discusses the result and presents related work and Section 7 concludes and draw some perspectives to be addressed in future work.

2. BACKGROUND

2.1 Kevoree at a glance

Kevoree [1] is an open-source dynamic component model, which relies on models at runtime [6] to properly support the dynamic adaptation of distributed systems. Models@runtime basically pushes the idea of reflection [23] one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance.

Kevoree has been influenced by previous work that we carried out in the DiVA project [23]. With Kevoree we push our vision of models@runtime [22] farther. In particular, Kevoree provides a proper support for distributed models@runtime. To this aim we introduce the *Node* concept to model the infrastructure topology and the *Group* concept to model semantics of inter node communication during synchronization of the reflection model among nodes. Kevoree includes a *Channel* concept to allow for multiple communication semantics between remote*Components* deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node, Group) obey the object type design pattern [18] to separate deployment artifacts from running artifacts. Kevoree supports multiple kinds of execution node technology (e.g. Java, Android, MiniCloud, FreeBSD, Arduino, ... [1]).

[1]http://www.kevoree.org

2.2 Dynamic Adaptation with Kevoree

Kevoree aims at providing advanced adaptation capabilities to different types of nodes:

- **Level 1: Parametric adaptation.** Dynamic update of parameter values, e.g. change of sampling rate in a component that wraps a physical sensor (adaptation of instance properties).
- **Level 2: Architectural adaptation.** Dynamic addition or removal of bindings or components, e.g. replication of software components and channels on different nodes to perform load balancing (adaptation of instances graph).
- **Level 3: Dynamic provisioning of types.** Hot deployment of component types that were not foreseen before the initial deployment of the system. This allows for system evolution by enabling parametric and architectural reconfigurations, including management of instances for types that are added and managed dynamically (adaptation of types).
- **Level 4: Adaptation for remote management.** Nodes supporting level 4 adaptation participate in a remote management layer, which supervises less powerful nodes. This layer monitors remote nodes by requesting their current Kevoree model; the layer triggers dynamic adaptation of nodes by sending precomputed reconfiguration scripts to them. This remote adaptation process supports seamless management of less powerful nodes by a more powerful one, which has enough resources to build and evaluate new and appropriate configurations.

The adaptation engine relies on a model comparison between two Kevoree models to compute a script for a safe system reconfiguration; execution of this script brings the system from its current configuration to the new selected configuration [23]. Model comparison yields a delta-model defining changes (using CRUD operations) that should be applied on the source model to obtain the target model. Planification algorithms [4] use this delta-model as input in order to defined an efficient schedule of the adaptation steps. The delta-model is finally compiled into a Kevoree script. The Kevoree Script language (KevScript for short) is a core language for describing reconfiguration. KevScript is comparable to FScript for Fractal Component Model [11]. Execution of a KevScript directly adapts a Kevoree system, without the need for a full Kevoree model definition. Such adaptation scripts are written by designers, or they can be generated by automated processes (*e.g.* within a control loop managing the Kevoree system).

3. MAPPING KEVOREE ADAPTATION CONCEPTS ON MICROCONTROLLERS

3.1 Challenges

Dynamic adaptation is a key concept to build advanced CPS able to adapt to their context and to user needs. Models at runtime is an efficient approach to manage the complexity of dynamic adaptation [23] by providing control and abstraction over reflection mechanisms. Applying reflection techniques is rather straightforward on fully grown component or service models such as OSGi, or directly on top of modern object-oriented languages such as Java, as long as

hal-00713769, version 1 - 2 Jul 2012

the execution hardware is powerful enough to run a virtual machine. The embedded software sensing and acting on the physical world (via hardware components) should be able to adapt to the needs of different (and potentially concurrent) services running on powerful nodes (in the cloud, on tablets, etc). Some services will for example subscribe to temperature alerts (the sensors being responsible to notify the service when a threshold has been reached), and then reconfigure the sensors to send almost continuous data, so that the service can precisely monitor the evolution of the temperature.

Applying models at runtime (or any dynamic adaptation technique) on execution nodes with scarce resources (e.g. microcontroller-based computation nodes) is much more difficult, for the following reasons:

1. **Downtime**: Microcontrollers often host the software that controls physical devices directly. Rebooting or freezing these microcontrollers may have severe consequences if the microcontrollers control safety critical devices, or unpleasant and noticeable effects if they control comfort devices.
2. **Volatile memory usage** (RAM): Dynamic memory allocation is the cornerstone that enables dynamic adaptation. Microcontrollers usually embed only of few kB of RAM, and this size limitations prohibits storing multiple configurations in memory at the same time.
3. **Persistent memory usage**: Persistent memory is required to ensure that the adaptation process has transaction-like properties, allowing recovery of microcontroller's state in case of a reboot after failure. EEPROM is a common type of persistent memory embedded into microcontrollers, usually with a very limited size. This type of memory also has a limited lifetime in term of numbers of writing operations. Similar to Solid State Disk [2], writes to EEPROM should be distributed among memory cells to optimize the lifetime of the overall memory.
4. **Recovery**: The ability to recover is critical for embedded systems, which are subject to failures (e.g. a temporary loss of power). Microcontrollers should reboot and restore their last configuration quickly enough to keep pace with configuration evolutions of the overall architecture.

CPS relying on a large set of autonomous sensors have cost and energy constraints that calls for cheap and power-efficient platforms able to run for long period of times with minimum on-site maintenance (*e.g.*, battery replacement). This is particularly true in the environmental monitoring domain: off-shore oil spills monitoring, flood prediction [22], air quality monitoring or radiation monitoring, for the following reasons[2]:

1. Their simplified architecture is robust and predictable: microcontrollers can operate by a wide range of temperature (typically -40 to 85 degrees Celcius), humidity, power supply, and have fixed number of cycles to execute a given operation.
2. Their energy needs (and generated heat) are very low: An 8-bit microcontroller running at 32kHz typically consumes less than 0,05W (less than 0,5W at 1MHz) excluding the need for any radiator. They can thus run for very long time on battery.
3. Their simplified architecture allows for mass production, making microcontrollers very cheap to deploy even in large numbers.

Compared to full-fledged computation nodes, cheap microcontrollers suffer from an adaptation overhead that stems from their hardware technology in terms of adaptation time or memory wear: dynamic provisioning of component types requires writing a program in flash memory. Therefore, implementing Kevoree concepts for microcontrollers nodes relies on a precise trade-off between flexibility and typical exploitation costs. Finding a lightweight solution for each of reconfiguration level described above is one of the main challenges of μ-kevoree.

3.2 Case study

We will use a smart building case study to validate Kevoree on a set of heterogeneous nodes, including of course some microcontrollers. Different systems (relying on proprietary devices and protocols) are usually deployed in buildings to manage different aspects of the building automation, in particular comfort (lighting, air conditioning, etc), safety and security (smoke and fire detection, sprinklers, etc). The degree of flexibility offered by a building automation system is often very poor.

- These systems rely on fixed topology of communication channels. Sensors and actuators often need to be physically coupled, hindering any future reconfiguration or evolution of the system. For example, a motion sensor will trigger all the lights of the corridor.
- The architecture is organized around a central server. When the devices are not physically coupled, they usually communicate through a central server, to execute the event-driven rules that orchestrate the system. Even though updating these rules is possible, to adapt the behavior of the system, this requires access to the central server.

The goal of Kevoree is to seamlessly distribute both the business logic and the dynamic adaptation capabilities on heterogeneous nodes ranging from powerful servers, to tablets, and to simple devices operated by microcontrollers. On a day-to-day basis, this would allow users to configure and reconfigure their offices on-the-fly from a smartphone, (e.g. to define a lighting environment according to the ambient luminosity, temperature, etc), while some other concerns would be managed by a central server (e.g. to turn the cameras on at night). In a crisis situation, this kind of seamless distribution would allow emergency services to cope with the failure of some nodes. Firemen could still access the data provided by low-level sensors, and compute meaningful context information on a tactical decision system despite the loss of a nodes.

4. DYNAMIC ADAPTATION FOR MICROCONTROLLERS

This section describes how Kevoree concepts are mapped to Arduino nodes. Arduino[3] is an open-source hardware and

[2]See for example http://www.atmel.com/Images/doc2545.pdf for detailed facts about microcontrollers

[3]http://www.arduino.cc

hal-00713769, version 1 - 2 Jul 2012

software electronics prototyping platform based on an 8-bits AVR microcontroller. Arduino boards can be connected to a set of sensors and actuators and programmed in languages from the C/C++ family. While we have chosen Arduino to implement our μ-Kevoree approach, it can be applied easily to other microcontroller families (PIC, ARM, etc).

Firmware implementations are often coded manually in C using a trial and error process, with an intensive manual and automated test-based validation. More advanced techniques (such as the MDE techniques proposed by ThingML[4] [15]) aim at generating static source code for microcontrollers. Such techniques can easily be leveraged to generate the internal code of component, which is not the scope of Kevoree. Microcontroller firmware puts a strong emphasis on resource usage (such as memory, CPU and energy needs) and reliability: microcontrollers can run for long periods of time and recover in case of power or connectivity loss. We acknowledge that these properties are critical, and the benefits provided by dynamic adaptation capabilities should not jeopardize them. Our work aims at breaking the static nature of code generation while preserving all benefits of low level code design.

To this aim our approach clearly separates structure and behavior: component type behaviors are currently implemented manually in C or in the Wiring language, using state of the art practice. One core contribution of our approach is the definition of a proper abstraction system to automate management of the adaptation logic of microcontrollers, by making their business logic a separate concept. Moreover, having a clear component structure with well defined inputs and outputs also eases testing at a more abstract level.

4.1 μ-Kevoree

This sub-section describes how the main concepts of Kevoree have been ported onto microcontrollers. μ-Kevoree is totally aligned and compatible with the exiting Java and Android versions.

Types. In Kevoree component types and channel types encapsulate business logic; they are generated as C structures. *Provided* ports of component types are mapped to methods, so that client components (which require ports of the same type) can invoke these methods and eventually push data. Kevoree properties that can be dynamically updated are simply mapped onto local variables contained by this structure. A local scheduler prevents concurrent calls on these variables. *Required* ports are generated as local structures, which can optionally refer to a bound channel instance. Similarly, channel types are generated as plain C structure. Outgoing channel bindings are generated as an internal array structure, enabling dynamic allocation and storage of external provided ports references.

Asynchronous message passing. As in Kevoree's implementations for Java and Android nodes, μ-Kevoree maps each port and channel onto an actor. More precisely, a FIFO queue is generated in front of each protected method. A dispatcher (local to each component) is then in charge of dispatching messages pushed on these queues to the correct method. This local scheduler is driven by a global scheduler described below.

Instance scheduler On each node a global instance scheduler is responsible for keeping its node in a consistent state by applying the following balance strategy:

- Periodic execution: the global scheduler periodically invokes the local scheduler of each component instance that has declared a periodic execution;
- Triggered execution: the global scheduler invokes the local scheduler of each component that has a non empty message queue.

The global scheduler also periodically checks for external messages related to dynamic adaptation, as described in the next two sub-sections.

4.2 Firmware flash to handle major evolutions

Flashing a microcontroller's firmware and then rebooting the device is an easy way to implement adaptation of a microcontroller node, by replacing the implementation entirely. This adaptation technique is acceptable in some specific and controlled contexts (initial production, on-site maintenance, etc), since the device is physically connected to a more powerful node using a communication link with broad bandwidth (e.g. wired link). In this case, flashing a controller's memory is rather safe (provided that the code of the firmware is safe) and also reasonably fast: flashing the entire memory by uploading the new firmware and then rebooting the device takes a few seconds only. However, this technique is problematic when the devices are deployed remotely. Flashing the firmware over-the-air is a hazardous manipulation: firmwares are typically bulk data (compared to other data usually transmitted on wireless links) and communication errors are more likely to occur; this requires advanced protocols to cope with error handling. In practice, this approach impacts significantly the time needed to install a new firmware.

Our approach limits flashing the full firmware to cases where new component types need to be deployed. In this regard C-based microcontrollers do not provide the same flexibility than Java/OSGi nodes with respect to dynamic provisioning and class loading. This is typically required for the initial deployment of the system where all the planned component types are provisioned, or for major evolutions of the system (e.g. to handle a new type of device not foreseen before the initial deployment). In all other cases such as reconfigurations of component instances for instance, our approach performs a partial flash memory update.

4.3 Seamless dynamic adaptation of microcontrollers

Following the principles of *model@runtime*, our dynamic adaptation process is fully automated, saving designers from writing low-level adaptation scripts or from entangling adaptation logic with business logic. Prior to any adaptation, all necessary checks on the new configuration are performed on the target model. Since microcontroller nodes have limited computational power, configuration checks are performed on more powerful, Java or Android based nodes. These checks aim at detecting a mismatch between the planned configuration and the physical hardware possibilities. After this validation step, the configuration is used as input for a generator algorithm, which computes a reconfiguration script. This script is then transmitted in a compact form to dependent microcontrollers. As communication errors are frequent in wireless sensors networks, we avoid problematic microcontroller states inconsistencies by implementing a roll-back based recovery mechanism.

[4] www.ThingML.net

hal-00713769, version 1 - 2 Jul 2012

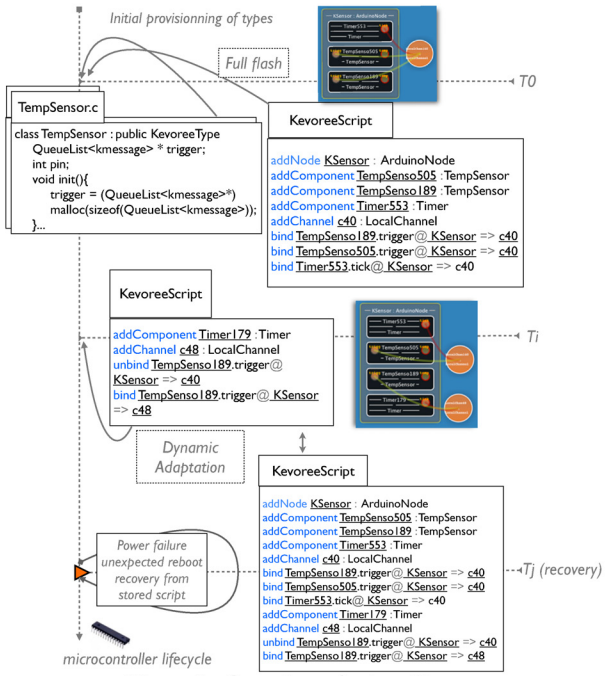


Figure 1: Overview of micro-Kevoree

The following table lists the dynamic adaptation levels supported by the different Kevoree node technologies; the levels are defined at the beginning of Section 2. The most powerful nodes are able to run Java programs and implement all levels of adaptation features. The more constrained are the nodes, the more tradeoffs have to be addressed.

Dynamic adaptation	JavaSE	Android	Arduino
Parametric	+	+	+
Architectural	+	+	+
Dynamic provision.	+	+	+/- firm. flash
Peers management	+	+/- perf. issues	- see Future work

While severely limited resource-wise, Arduino nodes are still able to support almost all dynamic adaptation features. In most cases, complex decisions will be taken by software running on more powerful nodes. Taking adaptation decisions require processing adaptation rules, optimize goals, etc, as microcontrollers usually do not have enough computational power.

Dynamic instantiation framework.

Reflection is a fundamental principle to achieve dynamic adaptation. But the C language has no support of the primitives required to build a full-fledged reflection model; this prevents a straightforward application of the model@runtime technique on microcontroller nodes. We have removed this limitation by generating code to emulate a reflection layer on these microcontrollers. We assume that the number of types is finite when generating the framework. Adding or removing a type then implies regeneration of the whole framework as described in Section 3. With this assumption of a closed type world, we generate a flat reflection layer by using exhaustive pattern matching on instances. We generate methods that take instances as parameter; these methods provide the following basic services:

Algorithm 1 μ-Kevoree core service

```
Function interruptScheduler(),resumeScheduler()
Function setProperty(instanceID,propID,propValue) : Bool
Function addBinding(channelID,componentID,portID) : Bool
Function removeBinding(channelID,componentID,portID) : Bool
Function createInstance(instanceID,typeID) : Bool
Function destroyInstance(instanceID) : Bool
Function exportCurrentState() : KevScript
```

We rely on script to export the state in order to optimize memory consumption. More precisely, every textual representation used to export the reflection model and related to type definitions (*e.g.*, port names) is locally encoded in static flash memory to save dynamic memory.

KevScript embedded interpretation.

An embedded KevScript interpreter uses the flattened reflexivity methods to implement basic services (e.g. instance life cycle, binding and parameters management).

Upon receiving a Kevoree script in a compressed format a microcontroller performs the following tasks:

Algorithm 2 KevScript Interpreter

```
Function interpretScript(script : KevScript)
  interruptScheduler()
  if permanentMemory.size >= PermanentMemoryLimit then
    resetPMemoryIndex()
    writeToPMemory(exportCurrentState())
  end if
  lastRecoveryPoint ← createRecoveryPointInPMemory()
  ∀ st, st ∈ script.statements → writeToPMemory(st)
  if executeFromPMemory(lastRecoveryPoint) then
    closeRecoveryPoint(lastRecoveryPoint)
  else
    rollback(lastRecoveryPoint)
  end if
  resumeScheduler()
```

Scripts are checked independently of the communication context and then stored. The new configuration is committed by an atomic write in the memory once it has been validated, thereby implementing an atomic transaction mechanism. This technique prevents incomplete memory saves. In case of any problems during the KevScript interpretation, a rollback is achieved by a simple reboot of the microcontroller. The Figure 1 gives an overview of this process taking as an example a temperature sensor reconfiguration. At T0 time a full configuration is pushed containing C code and an initial KevScript. At Ti time a KevScript is pushed adding 2 instances. Between Ti and Tj time a power failure occur resulting on a recovery using memory saved KevScript.

5. VALIDATION

This section describes our experimental setups for validating our approach against the criteria identified in Section 3[5]. More precisely, our experiments measure the following parameters:

Downtime: overall time needed by the microcontroller to adapt, including uploading of the new configuration. This metric thus measures the overhead induced by the microcontroller to manage its own state with respect to domain

[5]More details on this experiment can be found `http://blog.kevoree.org/pages/kevoree-for-microcontroller-benchmark`

hal-00713769, version 1 - 2 Jul 2012

applications execution time.
Volatile memory usage (RAM): amount of RAM memory dedicated to dynamic allocation of component instances, channels, and bindings. This metrics thus influences the maximum number of component instances, channels and bindings that a microcontroller can manage.
Persistent memory usage: amount of persistent memory used to store reconfiguration scripts and impact of storage strategy on memory life time. Persistent memory types such as the EEPROM embedded in the 8 bits AVR have a limited number of write cycles certified for each byte, thereby limiting the amount of storable data.
Recovery reboot delay: time needed by the microcontroller to reboot and restore its last configuration, after a crash or a loss of power.

We have used realistic configurations to assess our approach and evaluate the overhead induced by our dynamic component-based platform for microcontrollers, compared with static configurations that are updated by flashing the whole memory. All experiments were done using the Kevoree Arduino node implementation [6] running on an Arduino board with an ATMEL AVR 328P microcontroller. This processor embeds 32 KB of flash memory for storing programs, 2 KB of RAM memory and 1 KB of EEPROM. A flash-type memory (microSD) connected via an SPI bus was also used as persistent memory to assess the impact of memory type on results.

The following subsections show our specific experimental protocol and results, while the last subsection will present an industrial use case to validate the seamless integration of μKevoree devices in an existing dynamic architecture.

5.1 Downtime: How long does an adaptation freeze business logic?

Experimental setup. In this experiment we setup five different configurations, similar to the ones presented in the case study (building automation). The corresponding Kevoree models used different numbers of instances to simulate changes between the configuration used at night and a personalized configuration used during the day. More details on these models are available here [6] . In a nutshell, these models were configured with 4 nodes, hosting 0 to 10 instances each. Instances are implemented in C, with 30 lines of code each on average.

In a first step we generated the firmware of our test microcontroller, with code containing all type definitions used in this experiment. This step therefore includes code generation, compilation and writing into flash memory. Moreover, the generated code is automatically instrumented with probes to measure downtime and memory use (EEPROM and SDRAM). This step was repeated delayed of 100 ms with a new configuration that is chosen randomly; each new configuration was dynamically installed to replace the current running configuration. This random reconfiguration step was repeated 500 times. Figure 2 plots the raw data collected in this experiment. The plot on top shows that the RAM usage is constant. The second plot shows the downtime per reconfiguration, and third and bottom plots show downtime and script size respectively.

[6] `http://goo.gl/Xl2z9`

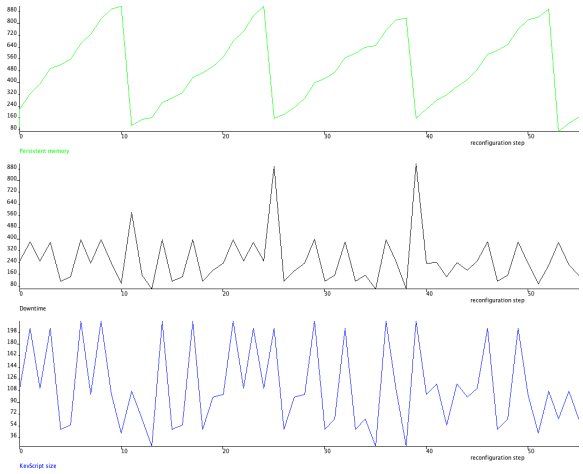


Figure 2: Experiment raw results[7]

Experimental results and analysis. Deploying a configuration by flashing the whole firmware is very costly: the downtime to deploy the initial configuration is 12.208 seconds. This high value comes mainly from the long transfer time of a full firmware but also from the time taken by the default boot loader to perform a full restart of the microcontroller. This value varies in a range of +/-2 seconds.

Results of this first experiment highlight that the size of the reconfiguration script is highly correlated with the downtime time: the Spearman correlation coefficient observed between script size and downtime is higher than 0.9. In addition, the compression algorithm used to decrease the script size in EEPROM has also an impact on downtime. We observed that the execution of this task is directly correlated with higher values of downtimes. This is discussed in the next subsection.

After 500 cycles of reconfiguration we measured the following extrema and mean values:

- minimum downtime of 58 ms, 210 (i.e. 12208 / 58) times faster than static flashing;
- maximum downtime of 916 ms, 14 (i.e. 12208 / 916) times faster than static flashing;
- mean downtime of 235 ms, 52 (12208 / 235) times faster than static flashing.

We used a distribution by percentiles graph for downtime values to better analyze these data, as shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
Downtime (ms)	58	59	139	221	248	398	916

The graph in Figure 3 clearly shows that the downtime values are clustered around 220 ms. 95% of the values are below 400 ms and 75% are below 250 ms. Then, only 5% of the values are above the 400 ms, which is explained by the EEPROM compression step. The lazy compression strategy allows us to limit the number of peaks and keep the maximum value around 200 ms. The highest values for the downtime are systematically linked to a reduction of the EEPROM size (caused by the compression routine). We observed 32 compressions of the EEPROM during the 500 reconfigurations *i.e.*, 6.4% of the reconfigurations trigger a compression so that they can be completely stored into the EEPROM. The maximum value of these 32 downtime peaks is 916 ms, the minimum is 218 ms and the mean value is 580.815 ms. This

hal-00713769, version 1 - 2 Jul 2012

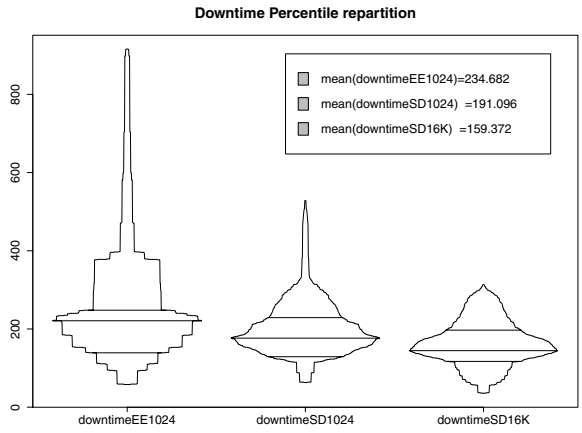


Figure 3: Flash RAM percentile downtime distribution (in ms)

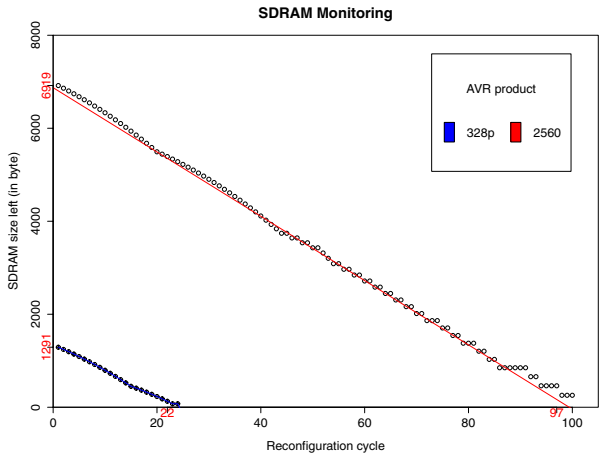


Figure 4: SDRAM capacity experiment

mean value is significantly higher than the mean value of the whole set of 500 reconfigurations (234.682 ms).

Probes also monitored SDRAM during this experiment. Neither memory leaks nor memory fragmentation occurred. Although the SDRAM is stable, it is necessary to check that the overhead induced by the framework actually allows for the dynamic creation of a realistically high number of instances, to match concrete use case needs. Our next experiment aimed at evaluating the capacity (in terms of dynamic instances) of our test microcontroller.

Experimental results and analysis with a different setup. We used the same protocol of 500 iterations, but we replaced the EEPROM with a 2 GB external flash memory (SD card), connected on an SPI bus. This experience is repeated twice: with only 1kb (same size as EEPROM), and with 16kb; results are shown in the following table.

Percentile(%)	0	5	25	50	75	95	100
DowntimeSD 1K(ms)	63	88	129	176	229	324	529
DowntimeSD 16K(ms)	35	56	117	145	197	297	314

Flash memory has a longer initialization time, which explains that the lowest values for the flash experiment are higher than the lowest value in the EEPROM experiment. However, writing speed of flash memory is high, resulting in a homogenization of downtime, which is under 200 ms most of the time. One can notice that the large increase of persistent memory (16 kb) clips the downtime peaks and therefore improves average downtime value. However this does not change the distribution of main values significantly.

5.2 Volatile memory usage: how many instances?

Experimental setup. The purpose of this experiment was to precisely determine the maximum number of instances that can fit into the SDRAM. An initial configuration was created with three instances: a timer, a switch and a default channel. Every 100 ms, the configuration was expanded by adding a new switch instance. Probes were injected to monitor the SDRAM.

Experimental results and analysis.
Figure 4 shows that SDRAM memory is full after 22 cycles, *i.e.* our test microcontroller can manage 25 instances (the 3 initial ones plus 22 additional instances). In practice, the count of devices controlled by one single microcontroller is approximately equal to the number of pins they have. In addition, microcontrollers should be able to run some code to orchestrate these devices and perform some computation. Kevoree is able to manage 25 instances (both for wrapping physical devices and for defining some orchestration) on our test microcontroller (22 pins). Despite a memory overhead, our approach is compatible with current practices.

Experiments results and analysis with a different setup. In this setup we used an ATMEL 2560 (4 KB of SDRAM) as our test microcontroller. The AVR 2560 accepts a load of 100 instances *i.e.* an improvement of+300% instances with +300% SDRAM. Again, the number of instances is larger than the number of pins (80) of the AVR 2560.

5.3 Persistent memory usage: How many certified reconfigurations?

Experimental setup. We used the setup of Section 5.1.

Experimental result and analysis. We observed that 500/32 = 15.625 reconfigurations can happen before the EEPROM (1 KB) is full, requiring a compaction using a new initial state. As for Solid State Disk [2], write operations to EEPROM should to be distributed throughout the memory in order to distribute wear. Our algorithm writes every byte before computing a new initial state. Each byte of this memory is certified for 100,000 writes [7]. Therefore if we assume 100 reconfigurations every day (which is much more than what is needed in most case studies), each byte of the EEPROM will be written 6.4 times a day on average, ensuring 15,625 days (i.e. about 43 years) of certified lifetime for the EEPROM.

Experiments results and analysis with a different setup.
In average, we can serialize the reconfiguration scripts of our experiment using 16 bytes. Since our algorithm ensures that every byte is written before the memory needs to be compressed, we can use the reasoning of the previous paragraph to compute the lifetime with a larger memory.

[7] http://arduino.cc/en/Reference/EEPROMWrite

hal-00713769, version 1 - 2 Jul 2012

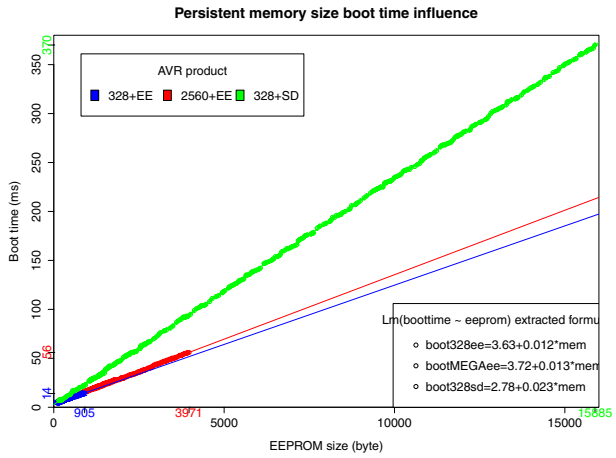


Figure 5: Persistent memory size boot time influence

5.4 Recovery reboot delay: How long to recover?

Experimental setup. This experiment used the configuration set described in Section 5.1, with only 50 cycles of reconfiguration performed every 2 seconds. The microcontroller was physically rebooted between each reconfiguration, and a new probe was inserted to measure the configuration restore time the after booting.

Experimental results and analysis.
Figure 5 displays the results of this second experiment. It appears that in the worst case with this AVR product (when the 1 KB EEPROM memory is almost full) the boot time is approximately 15 ms. In the best case (EEPROM almost empty) the boot time is approximately 3 to 4 ms. This value is mainly due to the slow read speed of the EEPROM embedded in the AVR. However, the time to restore a configuration is reasonable for most use cases, even in the worst case. We can therefore infer that the script size in EEPROM has a small impact on boot time with respect to save time. Therefore the best strategy is to use all available memory.

Experimental results and analysis with a different setup.
We used the same setup, but we replaced the EEPROM with a flash memory (1 KB and 16 KB). Using SD memory instead of EEPROM implies a longer boot time. We noticed a coefficient value of 0.012 for the EEPROM, and of 0.023 for the SD. This comes from the extra computation needed to read and write SD card. Unlike the EEPROM, the communication bus to access the SD flash is external to the microcontroller. The initialization time taken by flash memory configuration is linearly distributed to the limit of 16 KB. Above this size, the initialization time becomes greater than 360 ms. This is significantly higher than the reconfiguration time.

5.5 Discussion

The choice of persistent memory type and size depends on the use case needs. In some cases, there is a real need for traceability, and the history of the system should be kept *e.g.*, for *post mortem* analysis in case of failure. In other cases, performance of adaptation and boot time after a failure is more important. The benchmark developed in this approach can be useful to determine empirically which memory setup to use.

However, using an external memory significantly increases the price of such a platform. In addition, ensuring atomicity of reconfigurations is more difficult because of the asynchrony of transfer protocols. Existing protocols for interaction with SD cards (like MMC) are not suitable for storing reconfiguration scripts. More precisely, these scripts are much shorter (around 25 bytes in our experiments) than the minimum frame size (512 bytes) required by these protocols, and this leads to a significant overhead. In practice, most embedded devices combine EEPROM and flash memories. Kevoree allows designers to combine different memory types for different purposes.

μ-Kevoree overhead. Our framework adds several overhead sources, especially for the management of dynamic instance creation of components and channels. In order to quantify overhead, we measured volatile and program memory sizes on an HelloWorld program, using plain C and a Kevoree firmware setup on a 328P AVR microcontroller. The plain C version left 1842 free bytes after boot sequence, while the Kevoree version left 1604 bytes free. This represents an overhead about 11% of the total available RAM (242 bytes out of 2048). The plain C version used 2.3 KB of firmware memory, while the Kevoree use was 7.3 KB, giving an overhead of about 15% of the total 32kb available. The impact of Kevoree scheduler on processing cycles is highly program dependent, and we are currently experimenting further to compute this overhead.

Our synchronization and communication layer introduced an overhead under 15% on both memories, which is an acceptable value in the case of our IoT application. However, this impact should be evaluated in more depth for hard real-time applications, with a special attention to components needs in terms of processing cycles.

6. RELATED WORK

Software architecture aims at reducing complexity through abstraction and separation of concerns by providing a common understanding of component, connector and configuration [10, 21, 32]. One of the remaining challenges, strengthened by the Future Internet and CPS [24], is to properly manage dynamic architectures. SCA [8] is a standard that highlights modular software architecture concepts. It provides a model for composing applications that follow Service-Oriented Architecture principles. Frascati [29] is a SCA runtime that allows developing highly configurable applications. However, SCA focuses on rather heavy nodes typically able to run a JVM whereas μ-Kevoree also manages the dynamic adaptation of microcontroller-based systems, in addition to Java nodes, with a contained overhead.

Many approaches have highlighted the need for dynamic architectures to implement pervasive computing. A common approach consists in building a middleware to hide the heterogeneity of networks, hardware, operating systems, and programming languages.Rellermeyer *et al.* [26] for example provides an architecture for flexible interaction with electronic devices. Based on OSGi to implement a dynamic module system, their approach provides an abstraction layer

[8] `http://osoa.org/`

hal-00713769, version 1 - 2 Jul 2012

for device independence. Their architecture has the following non-functional benefits: scalability and ease of administration, flexibility, security, and efficiency. In a similar vein Escoffier *et al.* proposed iPOJO [13], a service component runtime that simplifies the development of OSGi applications. iPOJO has mainly been used in home-automation to implement service-oriented pervasive applications [7]. AutoHome is a middleware that extends the iPOJO component model, to create a framework to host autonomic home applications. Gaïa [27] is a CORBA-based meta-operating system for ubiquitous computing, built on top of a classical operating system aimint at abstracting the heterogeneity and complexity associated with ubiquitous environments. Olympus [25] proposes a high-level DSL to ease the development of Gaïa applications. Cassou *et al.* [9] proposes a generative programming approach to provide programming, execution and simulation support dedicated to the pervasive computing domain. They also demonstrate how abstraction can help to guide and verify the development of pervasive applications. Again, all these approaches rely on a reconfigurable middleware (often OSGi-based), and this restricts their deployment to powerful nodes, e.g. powerful enough to run a Java virtual machine. Our goal is to provide the same level of abstraction to develop pervasive and adaptive applications for powerful nodes and also for resource-constrained devices.

Several approaches have shown the benefits of using Model Driven Engineering (MDE) to design and reconfigure pervasive applications. Model-based approaches such as Matlab [9], Charon [3], UMLh [8], HyRoom [31], Masachio[16], Mechatronic UML [28], HyVisual [12], or SysML [10] propose a model to code development process with verification techniques to design modular embedded systems. However, none of these approaches support dynamic adaptation of a running system without first designing the adaptation at a business level. This, in practice, significantly reduces the number of configurations these approaches can manage. Indeed, these approaches provide no means to manage the combinatorial explosion of the number of configurations typically encountered in CPS: all configurations need to be explicitly designed.

Several approaches have shown the need of dynamic reconfiguration capabilities for embedded systems. Reconfigurable intelligent sensors are now able to confront major challenges in the design of cost-effective, energy-efficient, customizable systems, for example in the domain of health monitoring systems adaptable to individual users [19], or in the domain of operating system kernels [5]. Run-time reconfiguration can be achieved through programmable logic reconfiguration and/or software adaptation. In the first case, reconfigurable System on Chips [30] are a promising solution. To support software adaptation of embedded software, other works reuse a software architecture-based approach to the construction of embedded systems. For example, The Koala model [32], used for embedded software, allows late binding of reusable components with no additional overhead. Think [14] defines a component-based framework to support different mechanisms for dynamic reconfiguration and to select between them at build time, with no changes in operating system and application components. Different from these approaches, Fleurey *et al.* [15] present an approach based on state machines and an adaptation model to derive adaptive firmwares for microcontrollers. The approach relies on automatically enumerating configurations by exploring a set of adaptation rules defined at design time and compiling the resulting state-machine (which merges the business logic and the adaptation logic) into an optimized, yet static, firmware. In [17], Hofig *et al.* highlight the use of models@runtime for resource-constrained devices. They provide a UML state machine interpreter for AVR microcontrollers and compare the performance overhead with static code generation: model@runtime interpretation is adequate for the majority of situations, except when dealing with high-throughput or delay-sensitive data. Influenced by these approaches, Kevoree leverages models@runtime for microcontrollers and proposes new mechanisms to support dynamic reconfigurations and to select between them at runtime.

[9] `www.mathworks.com/products/matlab/`

[10] `http://www.sysml.org/`

7. CONCLUSION AND PERSPECTIVES

This paper presented μ-Kevoree, which pushes dynamicity and elasticity concerns directly into resource-constrained devices, based on the notion of models@runtime. This modeling layer that micro-controllers expose at runtime, enables the efficient and safe reasoning (by other Kevoree nodes: Java or Android) to adapt microcontroller-based nodes. In particular, this paper focused on the challenges met when mapping Kevoree and models@runtime features to low power microcontrollers, and on the required tradeoffs because of the stringent resource constraints.

This new version of Kevoree has been thoroughly evaluated with benchmarks in order to assess its usability in realistic setups. Despite an overhead with respect to static (non adaptive) code, these benchmarks have shown that 75% of the transactional reconfigurations can be performed in less than 250 ms, which is an acceptable value in many case studies. This is definitely faster (by a factor of almost 50) than a full memory rewrite of the firmware. Also, these benchmarks have shown that the time needed to reboot a microcontroller and restore its previous configuration is a linear function of the script size. For example, booting using a 1 kB EEPROM memory takes between 3 to 15 ms, while this memory size is large enough to store the script of 15 successive reconfigurations before needing compaction. Finally, the benchmarks have shown that Kevoree enables the deployment of software component instances in a number greater than the available pin count on the microcontroller. It is therefore possible to bind a software component to each physical device controlled by the microcontroller, and to deploy an extra component to coordinate these components.

In the future, we will improve the reliability of reconfigurations by making the computation of the initial state step transactional (compression of the persistent memory) *e.g.*, and by exploiting a circular rolling buffer on the persistent memory. Our scheduling algorithm is another area for improvement. Based on existing opportunistic garbage collectors (*e.g.* Java) we will leverage the computational cycles not used by hosted components to trigger the compression of the persistent memory in a lazy way, rather than waiting for a "memory full" event. We will also investigate further optimizations to reduce the memory consumption and the energy consumption. Finally we are planning to integrate simple reasoners in microcontrollers driven by μ-Kevoree so that they can operate in a fully autonomous mode, with no need to delegate the reasoning to a larger node.

hal-00713769, version 1 - 2 Jul 2012

Acknowledgment

This work has been funded by the EU FP7 projects: FI-PPP ENVIROFI, S-Cube NoE and MODERATES.

8. REFERENCES

[1] The Internet of Things Meets The Internet of People. `http://www.harborresearch.com/_literature_60961/The_Internet_of_Things_Meets_The_Internet_of_People`.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[3] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in charon. In *Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, HSCC '00, pages 6–19, London, UK, 2000. Springer-Verlag.

[4] F. André, E. Daubert, N. Grégory, B. Morin, and O. Barais. F4plan: An approach to build efficient adaptation plans. In *MobiQuitous*. ACM, 2010.

[5] S. Bagchi. Nano-kernel: a dynamically reconfigurable kernel for WSN. In *1st international conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, MOBILWARE '08, pages 10:1–10:6, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[6] G. S. Blair, N. Bencomo, and R. B. France. Models@run.time. *IEEE Computer*, 42(10):22–27, 2009.

[7] J. Bourcier, A. Diaconescu, P. Lalanda, and J. A. McCann. AutoHome: An Autonomic Management Framework for Pervasive Home Applications. *ACM Trans. Auton. Adapt. Syst.*, 6:8:1–8:10, February 2011.

[8] S. Burmester, H. Giese, and O. Oberschelp. Hybrid uml components for the design of complex self-optimizing mechatronic systems. In J. BRAZ, H. ARAÃŽJO, A. VIEIRA, and B. ENCARNAÃĞÃČO, editors, *INFORMATICS IN CONTROL, AUTOMATION AND ROBOTICS I*, pages 281–288. Springer Netherlands, 2006.

[9] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In *33rd International Conference on Software Engineering (ICSE'11)*, pages 431–440, Honolulu, US, 2011. ACM.

[10] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *24th International Conference on Software Engineering*, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM.

[11] P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. *Annales des Télécommunications*, 64(1-2):45–63, 2009.

[12] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[13] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *IEEE SCC*, pages 474–481. IEEE Computer Society, 2007.

[14] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think: A Software Framework for Component-based Operating System Kernels. In *General Track of the annual conference on USENIX Annual Technical Conference*, pages 73–86, Berkeley, CA, USA, 2002. USENIX.

[15] F. Fleurey, B. Morin, and A. Solberg. A Model-Driven Approach to Develop Adaptive Firmwares. In *SEAMS'11@ICSE: Workshop on Software Engineering for Adaptive and Self-Managing Systems*, Honolulu, Hawai, USA, 2011.

[16] T. Henzinger. Masaccio: A formal model for embedded components. In J. van Leeuwen, O. Watanabe, M. Hagiya, P. Mosses, and T. Ito, editors, *Theoretical Computer Science: Exploring New Frontiers of Theoretical Informatics*, volume 1872 of *Lecture Notes in Computer Science*, pages 549–563. Springer Berlin / Heidelberg, 2000.

[17] E. Höfig, P. H. Deussen, and I. Schieferdecker. On the performance of UML state machine interpretation at runtime. In *6th international symposium on Software engineering for adaptive and self-managing systems (SEAMS '11)*, pages 118–127, New York, USA, 2011. ACM.

[18] R. Johnson and B. Woolf. The Type Object Pattern, 1997.

[19] E. Jovanov, A. MilenkoviÄĞ, S. Basham, D. Clark, and D. Kelley. Reconfigurable Intelligent Sensors for Health Monitoring: A Case Study of Oximeter sensor. In *26th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4759–4762, 2004.

[20] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[21] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.

[22] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ Run.time to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[23] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[24] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[25] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A High-Level Programming Model for Pervasive Computing Environments. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 7–16, Washington, DC, USA, 2005. IEEE Computer Society.

[26] J. S. Rellermeyer, O. Riva, and G. Alonso. AlfredO: an architecture for flexible interaction with electronic devices. In *9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 22–41, New York, NY, USA, 2008. Springer-Verlag New York, Inc.

[27] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1:74–83, October 2002.

[28] W. Schäfer and H. Wehrheim. Graph transformations and model-driven engineering. chapter Model-driven development with Mechatronic UML, pages 533–554. Springer-Verlag, Berlin, Heidelberg, 2010.

[29] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 2011.

[30] H. Singh, M.-H. Lee, G. Lu, N. Bagherzadeh, F. J. Kurdahi, and E. M. C. Filho. Morphosys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Comput.*, 49:465–481, May 2000.

[31] T. Stauner, A. Pretschner, and I. Péter. Approaching a discrete-continuous uml: Tool support and formalization. In *Workshop of the pUML-Group on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*, pages 242–257. GI, 2001.

[32] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

Author manuscript, published in "Working IEEE/IFIP Conference on Software Architecture (2014)"

Scapegoat: an Adaptive monitoring framework for Component-based systems

Inti Gonzalez-Herrera
Univ Rennes 1 - IRISA
Campus de Beaulieu
35042 Rennes, France
inti.gonzalez_herrera@irisa.fr

Johann Bourcier
Univ Rennes 1 - IRISA
Campus de Beaulieu
35042 Rennes, France
johann.bourcier@irisa.fr

Erwan Daubert
Univ Rennes 1 - IRISA
Campus de Beaulieu
35042 Rennes, France
erwan.daubert@irisa.fr

Walter Rudametkin
INRIA
Campus de Beaulieu
35042 Rennes, France
walter.rudametkin@inria.fr

Olivier Barais
Univ Rennes 1 - IRISA
Campus de Beaulieu
35042 Rennes, France
barais@irisa.fr

François Fouquet
Univ du Luxembourg
Luxembourg, Luxembourg
francois.fouquet@uni.lu

Jean-Marc Jézéquel
Univ Rennes 1 - IRISA
Campus de Beaulieu
35042 Rennes, France
jezequel@irisa.fr

hal-00983045, version 1 - 24 Apr 2014

***Abstract*—Modern component frameworks support continuous deployment and simultaneous execution of multiple software components on top of the same virtual machine. However, isolation between the various components is limited. A faulty version of any one of the software components can compromise the whole system by consuming all available resources. In this paper, we address the problem of efficiently identifying faulty software components running simultaneously in a single virtual machine. Current solutions that perform permanent and extensive monitoring to detect anomalies induce high overhead on the system, and can, by themselves, make the system unstable. In this paper we present an optimistic adaptive monitoring system to determine the faulty components of an application. Suspected components are finely instrumented for deeper analysis by the monitoring system, but only when required. Unsuspected components are left untouched and execute normally. Thus, we perform localized *just-in-time* monitoring that decreases the accumulated overhead of the monitoring system. We evaluate our approach against a state-of-the-art monitoring system and show that our technique correctly detects faulty components, while reducing overhead by an average of 80%.**

I. Introduction

Modern computing systems, such as home automation, pervasive and ubiquitous systems are becoming a larger part of our lives. The tight connection with our living environment introduces new needs for these systems, such as the co-evolution of the system with its environment, the adaptation of the system to available resources and to users' behaviours, and the reliability of the system in front of faulty or malicious behaviours. Modern component frameworks assist software developers in coping with these new needs by providing introspection, reconfiguration, advanced technical services, among other facilities. These frameworks provide an extensible middleware and assist in managing technical issues such as security, transaction management, or distributed computing. They also support the simultaneous execution of multiple software components on the same virtual machine [34], [16], [11].

While component frameworks simplify the programming model for software developers, the isolation between the various components is limited because they are collocated on the same virtual machine. This allows components to be communicate efficiently and to share references to complex objects, something which is generally not possible when crossing the process boundary. However, one faulty software artifact may compromise the whole system by, for example, consuming all available resources on the machine. Furthermore, because these systems evolve in open environments where humans have central roles, software developers are unable to anticipate all future configurations of the application at design-time [3]. In these highly unpredictable environments, detecting irregular behaviour and maintaining the system in a consistent state is an important concern that can be addressed through continuous monitoring.

State of the art monitoring systems [17], [24], [7] extract steady data-flows of system parameters, such as the time spent executing a component, the amount of I/O and memory used, and the number of calls to a component. The overhead that these monitoring systems introduce into applications is high, which makes it unlikely for them to be used in production systems. Results presented in [8] show that overhead due to fine-grain monitoring systems can be up to a factor of 4.3. Our experiments, presented in this paper, show that overhead grows with the size of the monitored software. Thus, overhead greatly limits the scalability and usage of monitoring systems.

In this paper, we address excessive overhead in monitoring approaches by introducing an optimistic adaptive monitoring system that provides lightweight global monitoring under normal conditions, and precise and localized monitoring when problems are detected. Although our approach reduces the accumulated amount of overhead in the system, it also introduces a delay in finding the source of a faulty behaviour. Our objective is to provide an acceptable trade-off between the overhead and the delay to identify the source of faulty behaviour in the system.

Our optimistic adaptive monitoring system is based on the following principles:

hal-00983045, version 1 - 24 Apr 2014

- **Contract-based resource usage.** The monitoring system follows component-based software engineering principles. Each component is augmented with a contract that specifies their expected or previously calculated resource usage [5]. The contracts specify how a component uses memory, I/O and CPU resources.
- **Localized just-in-time injection and activation of monitoring probes.** Under normal conditions our monitoring system performs a lightweight global monitoring of the system. When a problem is detected at the global level, our system activates local monitoring probes on specific components in order to identify the source of the faulty behaviour. The probes are specifically synthesized according to the component's contract to limit their overhead. Thus, only the required data are monitored (e.g., only memory usage is monitored when a memory problem is detected), and only when needed.
- **Heuristic-guided search of the problem source.** We use a heuristic to reduce the delay of locating a faulty component as quickly as possible while maintaining an acceptable overhead. This heuristic is used to inject and activate monitoring probes on the suspected components. However, overhead and latency in finding the faulty component are greatly impacted by the precision of the heuristic. A heuristic that quickly locates faulty components will reduce both delays and the accumulated overhead of the monitoring system. We propose using Models@run.time techniques in order to build an efficient heuristic.

The evaluation of our optimistic adaptive monitoring system shows that, in comparison to other state-of-the-art approaches, the overhead of the monitoring system is reduced by up to 80%. Regarding latency, our heuristic reduces the delay to identify the faulty component when changing from global, lightweight monitoring to localized, just-in-time monitoring.

The remainder of this paper is organized as follows. Section II presents the background on Models@run.time and motivates our work through a case study which is used to validate the approach. Section III provides an overview of the Scapegoat framework. It highlights how the component contracts are specified, how monitoring probes are injected and activated on-demand, how the ScapeGoat framework enables the definition of heuristics to detect faulty components without activating all the probes, and how we benefit from Models@run.time to build efficient heuristics. Section IV validates the approach through a comparison of detection precision and detection speed with other approaches. Finally, section V discusses related work and section VI discusses the approach and presents our conclusion and future work.

II. Background and motivating example

A. Motivating example

In this section we present a motivating example for the use of an optimistic adaptive monitoring process in the context of a real-time crisis management system in a fire department. During a dangerous event, many firefighters are present and need to collaborate to achieve common goals. Firefighters have to coordinate among themselves and commanding officers need to have an accurate real-time view of the system.

The Daum project[1] provides a software application that supports firefighters in these situations. The application runs on devices with limited computational resources because it must be mobile and taken on-site. It provides numerous services for firefighters depending on their role in the crisis. In this paper we focus on the two following roles:

- A collaborative functionality that allows commanding officers to follow and edit tactical operations. The firefighters' equipment include communicating sensors that report on their current conditions.
- A drone control system which automatically launches a drone equipped with sensors and a camera to provide a different point-of-view on the current situation.

As is common in many software applications, the firefighter application may have a potentially infinite number of configurations. These configurations depend on the number of firefighters involved, the type of crisis, the available devices and equipment, among other parameters. Thus, it is generally not possible to test all configurations to guarantee that the software will always function properly. Consequently, instead of testing all configurations, there is a need to monitor the software's execution to detect faulty behaviours and prevent system crashes. However, fine-grained monitoring of the application can have excessive overhead that makes it unsuitable with the application and the devices used in our example. Thus, there is a need for an accurate monitoring system that can find faulty components while reducing overhead.

The Daum project has implemented the firefighter application using a Component Based Software Architecture. The application makes extensive use of the Kevoree[2] component model and runtime presented below.

B. Kevoree

Kevoree is an open-source dynamic component platform, which relies on Models@run.time [10] to properly support the dynamic adaptation of distributed systems. Our use case application and the implementation of the Scapegoat framework make extensive use of the Kevoree framework. The following subsections detail the background on component-based software architecture, introduce the Models@run.time paradigm and give an overview of the Kevoree platform.

1) Component-based software architecture: Software architecture aims at reducing complexity through abstraction and separation of concerns by providing a common understanding of component, connector and configuration [15], [28], [35]. One of the benefits is that it facilitates the management of dynamic architectures, which becomes a primary concern in the Future Internet and Cyber-Physical Systems [31]. Such systems demand techniques that let software react to changes by self-organizing its structure and self-adapting its behaviour. Many works [20] have shown the benefits of using component-based approaches in such open-world environments [3].

To satisfy the needs for adaptation, several component models provide solutions to dynamically reconfigure a software architecture through, for example, the deployment of new

[1] https://github.com/daumproject
[2] http://www.kevoree.org

hal-00983045, version 1 - 24 Apr 2014

modules, the instantiation of new services, and the creation of new bindings between components. In practice, component-based (and/or service-based) platforms like Fractal [11], OpenCOM [12], OSGi [34] or SCA [32] provide platform mechanisms to support dynamic architectures.

2) Models@run.time: Built on top of dynamic component frameworks, Models@run.time denote model-driven approaches that aim at taming the complexity of dynamic adaptation. It basically pushes the idea of reflection [30] one step further by considering the reflection-layer as a real model: "something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality". In practice, component-based and service-based platforms offer reflection APIs that allow instrospecting the application (e.g., which components and bindings are currently in place in the system) and dynamic adaptation (e.g., changing the current components and bindings). While some of these platforms offer rollback mechanisms to recover after an erroneous adaptation [25], the purpose of Models@run.time is to prevent the system from actually enacting an erroneous adaptation. In other words, the "model at runtime" is a reflection model that can be decoupled from the application (for reasoning, validation, and simulation purposes) and then automatically resynchronized. This model can not only manage the application's structural information (i.e., the architecture), but can also be populated with behavioural information from the specification or the runtime monitoring data.

The Kevoree framework: Kevoree provides multiple concepts that are used to create a distributed application that allows dynamic adaptation. The *Node* concept is used to model the infrastructure topology and the *Group* concept is used to model the semantics of inter-node communication, particularly when synchronizing the reflection model among nodes. Kevoree includes a *Channel* concept to allow for different communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (*Component*, *Channel*, *Node*, *Group*) obey the object type design pattern [21] in order to separate deployment artifacts from running artifacts.

Kevoree supports multiple execution platforms (e.g., Java, Android, MiniCloud, FreeBSD, Arduino). For each target platform it provides a specific runtime container. Moreover, Kevoree comes with a set of tools for building dynamic applications (a graphical editor to visualize and edit configurations, a textual language to express reconfigurations, several checkers to valid configurations).

As a result, Kevoree provides a promising environment by facilitating the implementation of dynamically reconfigurable applications in the context of an open-world environment. Because our goal is to design and implement an adaptive monitoring system, the introspection and the dynamic reconfiguration facilities offered by Kevoree suit the needs of the ScapeGoat framework.

III. The ScapeGoat framework

Our optimistic adaptive monitoring system extends the Kevoree platform with the following principles: i) component contracts that define per-component resource usage, ii) localized and just-in-time injection and activation of monitoring probes, iii) heuristic-guided faulty component detection. The following subsections present an overview of these three principles in action.

A. Specifying component contracts

In both the Kevoree and ScapeGoat approaches, we follow the contract-aware component classification [5], which applies B. Meyer's Design-by-Contract principles [29] to components. In fact, ScapeGoat provides Kevoree with *Quality of Service* contract extensions that specify the worst-case values of the resources the component uses. The resources specified are memory, CPU, I/O and the time to service a request.

For example, for a simple Web server component we can define a contract on the number of instructions per second it may execute [7] and the maximum amount of memory it can consume. The number of messages can be specified per component or per component-port. The example is shown in Listing 1.[3] This contract extension follows the component interface principle [1], and allows us to detect if the problem comes from the component implementation or from a component interaction. That is, we can distinguish between a component that is using excessive resources because it is faulty, or because other components are calling it excessively.

```
addComponent WsServer650@node0 : WsServer {
  // Specify that this component can use 258 CPU
  // instructions per second,
  cpu_wall_time = '258',
  // Specify that this component can consume a maximum of 15000
  // bytes of memory,
  memory_max_size = '15000',
  // Specify that the contract is guaranteed under the assumption that
  // we do not receive more than 10k messages on the component and
  // 10k messages on the port named service
  // ( this component has only one port )
  throughput_msg_per_second='all=10000;service=10000'
}
```

Listing 1. Component contract specification example

B. An adaptive monitoring framework within the container

Scapegoat provides a monitoring framework that adapts its overhead to current execution conditions and leverages the architectural information provided by Kevoree to guide the search for faulty components. The monitoring mechanism is mainly injected within the component container.

Each Kevoree node/container is in charge of managing the component's execution and adaptation. Following the Models@run.time approach, each node can be sent a new architecture model that corresponds to a system evolution. In this case, the node compares its current configuration with the configuration required by the new architectural model and computes the list of individual adaptations it must perform. Among these adaptations, the node is in charge of downloading all the component packages and their dependencies, and loading them into memory. During this process, Scapegoat provides the existing container with (i) checks to verify that the system has enough resources to manage the new component, and (ii) instrumentation for the component's classes in order

[3] Contract examples for the architecture presented in section II-A can be found at http://goo.gl/uCZ2Mv.

hal-00983045, version 1 - 24 Apr 2014

to add bytecode for the monitoring probes. Scapegoat uses the components' contracts to check if the new configuration will not exceed the amount of resources available on the device. And Scapegoat instruments the components' bytecode to monitor object creation (to compute memory usage), to compute each statement (for calculating CPU usage), and to monitor calls to classes that wrap I/O access such as the network or file-system.

We provide several instrumentation levels that vary in the information they obtain and in the degree they impact the application's performance:

- **Global monitoring** does not instrument any components, it simply uses information provided directly by the JVM.
- **Memory instrumentation** or memory accounting, which monitors the components' memory usage.
- **Instruction instrumentation** or instruction accounting, which monitors the number of instructions executed by the components.
- **Memory and instruction instrumentation**, which monitors both memory usage and the number of instructions executed.

Probes are synthesized according to the components' contracts. For example, a component whose contract does not specify I/O usage will not be instrumented for I/O resource monitoring. Probes can be dynamically activated or deactivated, with the exception of memory usage probes. Memory consumption probes must remain activated to guarantee that all memory usage is properly accounted for, from the component's creation to the component's destruction. Indeed, deactivating memory probes would cause object allocations to remain unaccounted for. However, probes for CPU and I/O usage can be activated on-demand to check for component contract compliance.

To take advantage of having dynamic probes, we attempt to minimize the overhead of the monitoring system by activating selected probes only when a problem is detected at the global level. We estimate the most likely faulty components and then activate the pertinent monitoring probes. This technique means we only activate fine-grain monitoring on components suspected of misbehavior. After monitoring this subset of components, if any of them are indeed faulty, the monitoring system has finished and determines that these components are the source of the problem. If the subset of components are determined to be healthy, the system continues its search and starts monitoring the second most likely faulty subset. The monitoring mechanism implemented for ScapeGoat is summarized in listing 2.

As a result, at any given moment, applications must be in one of the following monitoring modes:

- **No monitoring.** The software is executed without any monitoring probes or modifications.
- **Global monitoring.** Only global resource usage is being monitored, such as the CPU usage and memory usage at the Java Virtual Machine (JVM) level.
- **Full monitoring.** All components are being monitored for all types of resource usage. This is equivalent to current state-of-the-art approaches.

```
monitor(C: Set<Component>, heuristic : Set<Component>→Set<Component>)
  init memory probes (c | c ∈ C ∧ c.memory_contract ≠ ∅)
 while container is running
   wait violation in global monitoring
   checked = ∅
   faulty = ∅
   while checked ≠ C ∧ faulty = ∅
     subsetToCheck = heuristic ( C \ checked )
     instrument for adding probes ( subsetToCheck )
     faulty = fine−grain monitoring( subsetToCheck )
     instrument for removing probes ( subsetToCheck )
     checked = checked ∪ subsetToCheck
   if faulty ≠ ∅
     adapt the system ( faulty , C)

fine−grain monitoring( C : Set<Component> )
  wait few milliseconds // to obtain good information
  faulty = {c | c ∈ C ∧ c.consumption > c.contract}
  return faulty
```

Listing 2. The main monitoring loop implemented in ScapeGoat

- **Localized monitoring.** Only a subset of the components are monitored.
- **Adaptive monitoring.** The monitoring system changes from Global monitoring to Full or Localized monitoring if a faulty behaviour is detected.

For the rest of this paper we use the term **all components** for the adaptive monitoring policy that indicates that the system changes from *global monitoring* mode to *full monitoring* mode if and when a faulty behaviour is detected.

1) ScapeGoat's architecture: The Scapegoat framework is built using the Kevoree component framework. Scapegoat extends Kevoree by providing a new Node Type and three new Component Types:

- **Monitored Node.** Handles the admission of new components by storing information about resource availability. Before admission, it checks the security policies and registers components with a contract in the monitoring framework. Moreover, it intercepts and wraps class loading mechanisms to record a component type's loaded classes. Such information is used later to (de)activate the probes.
- **Monitoring Component.** This component type is in charge of checking component contracts. Basically, it implements a complex variant of the algorithm in listing 2. It communicates with other components to identify suspected components.
- **Ranking Component.** This is an abstract Component Type; therefore it is user customizable. It is in charge of implementing the heuristic that ranks the components from the most likely to be faulty to the least likely.
- **Adaptation component.** This component type is in charge of dealing with the adaptation of the application when a contract violation is detected. It is also a customizable component. The adaptation strategy whenever a faulty component is discovered is out os scope of this paper. Nevertheless, several strategies may be implemented in Scapegoat, such as removing faulty components or slowing down communication between components when the failure is due to a violation in the way one component is using another.

hal-00983045, version 1 - 24 Apr 2014

2) Implementation strategy: Scapegoat aims minimizing monitoring overhead when the framework is running in *Global Monitoring* mode. To achieve this, ScapeGoat removes as many monitoring probes as possible and only activates probes that are required. This requires changing the bytecode that defines the application's classes at runtime, when the monitoring mode changes. Bytecode is changed at a per-component basis. We use the ASM library to perform bytecode manipulation and a Java agent to get access to and transform the classes. Bytecode manipulation has been proposed before for resource accounting and profiling in Java [6], [7], [14].

C. Leveraging Models@run.time to build an efficient monitoring framework

As presented in section III-B, our approach offers a dynamic way to activate and deactivate fine-grain localized monitoring. We use a heuristic to determine which components are more likely to be faulty. Suspected components are the first to be monitored.

Our framework can support different heuristics, which can be application or domain-specific. In this paper we propose a heuristic that leverages the use of the Models@run.time approach to infer the faulty components. The heuristic is based on the assumption that the cause of newly detected misbehavior in an application is likely to come from the most recent changes in the application. This can be better understood as follows:

- recently added or updated components are more likely to be the source of a faulty behaviour;
- components that directly interact with recently added or updated components are also suspected.

We argue that when a problem is detected it is probable that recent changes have led to this problem, or else, it would have likely occurred earlier. If recently changed components are monitored and determined to be healthy, it is probable that the problem comes from direct interactions with those components. Indeed, changes to interactions can reveal dormant issues with the components. The algorithm used for ranking the components is presented in more detail in listing 3. In practice, we leverage the architectural-based history of evolutions of the application, which is provided by the Models@run.time approach.

```
ranker () : list <Component>
  visited = ∅
  ranking = {}
  for each model M ∈ History
    N = {c | c was added in M}
    Neighbors = ⋃_{c∈N} c.neighbors
    ranking.add N\visited
    visited = visited ∪ N
    SortedNeighbors = sort (Neighbors \ visited )
    ranking.add SortedNeighbors
    visited = visited ∪ Neighbors
  return ranking

sort (S : Set<Component>) : list<Component>
  r = {}
  if S ≠ ∅
    choose b | b ∈ S ∧ b is newer than any other element in S
    r.add b, sort (S\{b})
  return r
```

Listing 3. The ranking algorithm (uses the model history for ranking).

IV. Evaluation

In this section we present our experiments and discuss the usability of our approach. We focus on the following research questions to assess the quality and the efficiency of ScapeGoat:

- **What is the impact of the various levels of instrumentation on the application?** Our approach assumes high overhead for full monitoring and low overhead for a lightweight global monitoring system. The experiments presented in section IV-A show the overhead for each instrumentation level.
- **Does our adaptive monitoring approach have better performance than state-of-the-art monitoring solutions?** The experiment presented in section IV-B highlights the performances benefits of our approach considering a real-world scenario.
- **What is the impact of using a heuristic in our adaptive monitoring approach?** The experiment presented in section IV-C highlights the impact of the application and component sizes, and the need of a good heuristic to quickly identify faulty components.

The efficiency of our monitoring solution is evaluated on two dimensions: the overhead on the system and the delay to detect failures. We show there is a trade-off between the two dimensions and that ScapeGoat provides a valuable solution that increases the delay to detect a faulty component but reduces accumulated overhead.

a) Use case: We have built several use cases based on a template application from our motivating example in section II-A. We reused an open-source crisis-management application for firefighters that has been built with Kevoree components. We use two functionalities of the crisis-management application. The first one is for managing firefighters. The equipment given to each firefighter contains a set of sensors that provides data for the firefighter's current location, his heartbeat, his body temperature, his acceleration movements, the environmental temperature, and the concentration of toxic gases. These data are collected and displayed in the crisis-management application, which provides a global-view of the situation. The second functionality uses drones to capture real-time video from an advantageous point-of-view.

Figure 1 shows the set of components that are involved in our use-case, including components for firefighters, drones and the crisis-management application[4]. The components in the crisis-management application are used in our experiments, but the physical devices (drones and sensors) are simulated through the use of mock components.

Every use case we present extends the crisis-management base application by any one of the following possibilities: adding new or redundant components, adding external Java applications with wrapper components (e.g., Weka, DaCapo), or modifying existing components (e.g., to introduce a fault into them).

[4]More information about these components is given in http://goo.gl/x64wHG

hal-00983045, version 1 - 24 Apr 2014

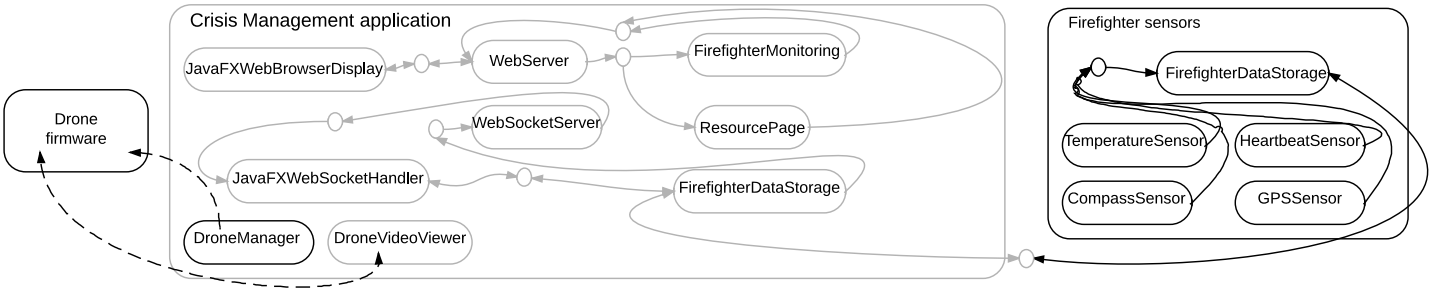


Fig. 1. The component configuration for our crisis-management use-case.

b) Measurement Methodology: To obtain comparable and reproducible results, we used the same hardware across all experiments: a laptop with a 2.90GHz Intel(R) i7-3520M processor, running Fedora 19 with a 32 bit kernel and 8GB of system memory. We used the HotSpot Java Virtual Machine version 1.7.0-40, and Kevoree framework version 2.0.12. Each measurement presented in the experiment is the average of ten different runs under the same conditions.

The evaluation of our approach is tightly coupled with the quality of the resource consumption contracts attached to each component. We built the contracts following classic profiling techniques. The contracts were built by performing several runs of our use cases, without inserting any faulty components into the execution. Firstly, we executed the use cases in an environment with global monitoring activated to get information for the global contract. Secondly, per-component contracts were created by running the use cases in an environment with full monitoring.

A. Overhead of the instrumentation solution

Our first experiment compares the various instrumentation levels to show the overhead of each one. In this experiment, we compare the following instrumentation levels: *No monitoring*, *Global monitoring*, *Memory instrumentation*, *Instructions instrumentation*, *Memory and instructions instrumentation* (i.e., Full monitoring).

In this set of experiments we used the DaCapo 2006 benchmark suite [9]. We developed a Kevoree component to execute this benchmark [5]. The container was configured to use full monitoring and the parameters in the contract are upper bounds of the real consumption[6].

Figure 2 shows the execution time of several DaCapo tests under different scenarios. First, we wish to highlight that Global monitoring introduces no overhead compared with the *No monitoring* mode. Second, the overhead due to memory accounting is lower than the overhead due to instruction accounting. This is very important because, as we described in section III-B, memory probes cannot be deactivated dynamically. We calculated the overhead as:

$$overhead = \frac{WithInstrumentation}{GlobalMonitoring}$$

[5]http://goo.gl/V5T6De

[6]Scripts are generated from those available at http://goo.gl/FR8LC7.

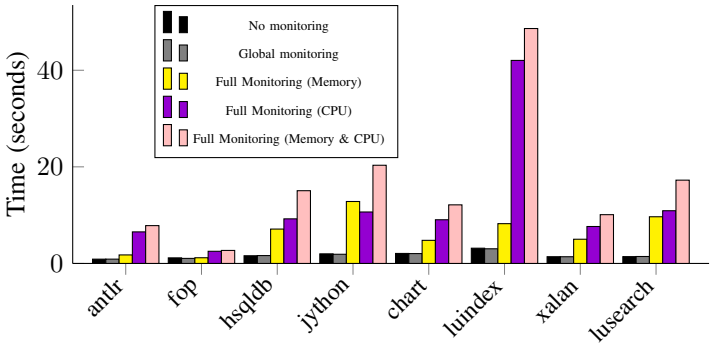


Fig. 2. Execution time for tests using the DaCapo Benchmark

The average overhead due to memory accounting is 3.70, while the value for instruction accounting is 6.54. These values are not as good as the values reported in [8]; the difference is negligible for memory accounting but they obtain a superior value between 3.2 and 4.3 for instructions accounting. The performance difference comes from a specific optimization that we chose not to apply. The optimization provides fast access to the execution context by adding a new parameter to each method. Nevertheless, this solution needs to keep a version of the method without the new parameter because native calls cannot be instrumented like that. We decided to avoid such an optimization because duplication of methods increases the size of the applications, and with it, the memory used by the heap. In short, our solution can reach similar values if we include the mentioned optimization, but at the cost of using more memory. On the other hand, the values we report are far lower than the values reported in [8] for hprof. Hence, we consider that our solution is comparable to state of the art approaches in the literature.

In addition, we plan to study alternatives to improve instruction accounting. For example, we plan to study the use of machine learning for monitoring [33]. Based on a machine learning approach, it is possible to train the monitoring system to do the instruction instrumentation. Then, instead of doing normal instruction instrumentation, we might only do, for example, method-calls instrumentation and with the learning data, the monitoring system should be able to infer the CPU usage of each call, whilst lowering the overhead.

The results of our experiment shown in figure 2 demonstrate the extensive impact the *Full monitoring* mode, which

hal-00983045, version 1 - 24 Apr 2014

uses either *Memory instrumentation* or *CPU instrumentation*, has on the application. Our *Adaptive monitoring* mode, which uses *Global monitoring* and switches to *Full monitoring* or *localized monitoring*, is able to reduce this accumulated overhead due to the fact that *Global monitoring* has no appreciable overhead.

B. Overhead of Adaptive Monitoring vs Full Monitoring

The previous experiment highlights the usefulness of using *Adaptive monitoring*. However, switching from *Global monitoring* to either *Full* or *Localized monitoring* introduces an additional overhead due to having to instrument components and activate monitoring probes. Our second experiment compares the overhead introduced by the adaptive monitoring with the overhead of *Full monitoring* as used in state-of-the-art monitoring approaches.

Table I shows the tests we built for the experiment. We developed the tests by extending the template application. Faults were introduced by modifying an existing component to break compliance with its resource consumption contract. We reproduce each execution repetitively; thus, the faulty behaviour is triggered many times during the execution of the application. The application is not restarted.

Figure 3 shows the execution time of running the use cases with different scenarios. Each scenario uses a specific monitoring policy (*Full monitoring*, *Adaptive monitoring with All Components*, *Adaptive monitoring with Localized monitoring*, *Global monitoring*). This figure shows that the overhead differences between *Full monitoring* and *Adaptive monitoring with All Components* is clearly impacted by scenarios that cause the system to transition too frequently between a lightweight Global and a fine-grain Adaptive monitoring. Such is the case for use cases UC3 and UC4 because the faulty component is inserted and never removed. Using *Adaptive monitoring* is beneficial if the overhead of *Global monitoring* plus the overhead of switching back and forth to *All Components monitoring* is less than the overhead of the *Full monitoring* for the same execution period. If the application switches between monitoring modes too often then the benefits of adaptive monitoring are lost.

The overhead of switching from *Global monitoring* to *full components* or *Localized monitoring* comes from the fact that the framework must reload and instrument classes to activate the monitoring probes. Therefore, using *Localized monitoring* reduces the number of classes that must be reloaded. This is shown in the third use-case of figure 3, which uses a heuristic based on the number of failures. Because we execute the faulty component many times, the heuristic is able to select, monitor and identify the faulty component quickly. This reduces overhead by 80%. We use the following equation to calculate overhead:

$$Gain = 100 - \frac{OurApproach - GlobalMonitoring}{FullMonitoring - GlobalMonitoring} * 100$$

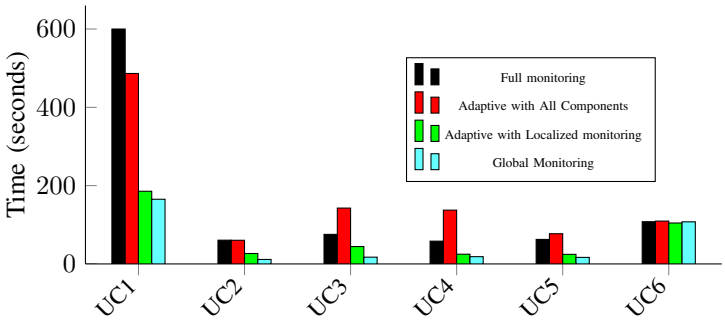


Fig. 3. Execution time for some use cases under different monitoring policies.

C. Overhead from switching monitoring modes, and the need of a good heuristic

As we explain in the previous experiments, even if using *Localized monitoring* is able to reduce the overhead of the monitoring system, the switch between *Global* and *Localized monitoring* introduces additional overhead. If this overhead is too high, the benefits of adaptive monitoring are lost.

In this experiment we show the impact of the application's size, in terms of number of components, and the impact of the component's size, in terms of number of classes, on adaptive monitoring. We also show that the choice of the heuristic to select suspected components for monitoring is important to minimize the overhead caused from repeated instrumentation and probe activation processes.

For the use case, we created two components and we introduced them into the template application separately. Both components perform the same task, which is performing a primality test on a random number and sending the number to another component. However, one of the components causes 115 classes to be loaded, while the other only loads 4 classes.

We used the same basic scenario with a varying number of primality testing components and component sizes. In this way, we were able to simulate the two dimensions of application size. The exact settings, leading to 12 experiments, are defined by the composition of the following constraints:

- $N_{comp} = \{4, 8, 16, 32, 64, 128\}$ which defines the number of components for the application
- $Size_{comp} = \{4, 115\}$ which defines the number of classes for a component

With these use cases, we measured the delay to find the faulty component and the execution-time overhead caused by monitoring. Figures 4 and 5 show the delay to detect the faulty component with regards to the size of the application. In the first figure, the component size is 115 classes, and in the second figure, the component size is four classes.

1) Impact of the application size: Using figures 5 and 7, we see that the size of the application has an impact on the delay to detect faulty components, and also on the monitoring overhead. We also calculated the time needed to find the faulty component with the *All components* mode after its initialization (the time needed to switch from *Global monitoring*). This time is around 2 seconds no matter the size of the application.

hal-00983045, version 1 - 24 Apr 2014

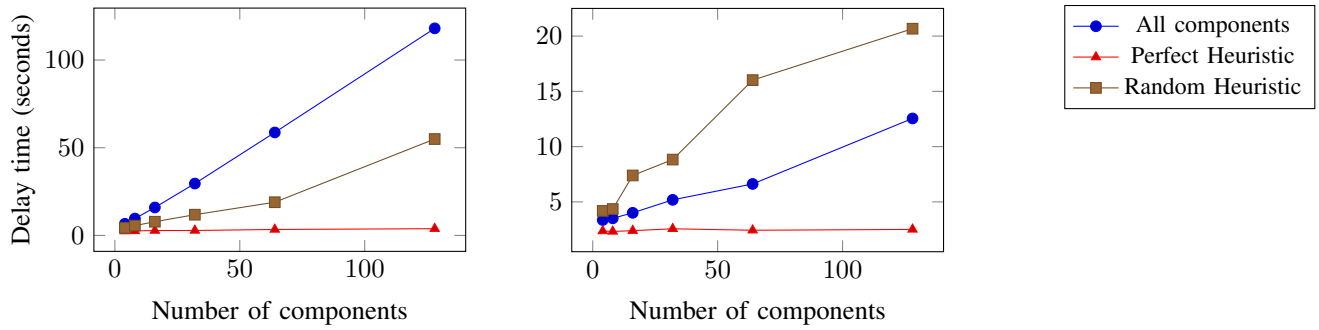


Fig. 4. Delay time to detect fault with a component size of 115 classes.

Fig. 5. Delay time to detect fault with a component size of four classes.

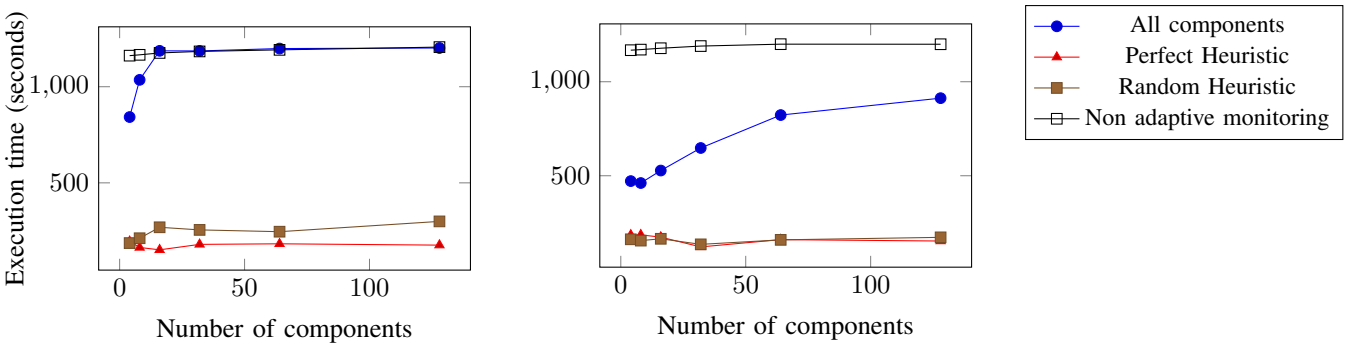


Fig. 6. Execution time of main task with a component size of 115 classes.

Fig. 7. Execution time of main task with a component size of four classes.

That is the reason the switch from *Global monitoring* to *All components* has such a large effect on overhead.

These figures also show that using *Localized monitoring* instead of *All components* when switching from *Global monitoring* helps reduce the impact of the application's size by reducing the number of components to monitor and the number of classes to instrument. However, we also see that using a sub-optimal heuristic may have negatively impacted the delay to detect faulty components. This can be explained by the multiple switches that the Random heuristic may often require to locate the faulty component.

2) Impact of the component size: In figures 4 and 5 we see that even if we use a good heuristic, the size of the components to monitor impacts the delay in detecting the faulty component. Like the application's size, component size impacts the switch from *Global monitoring* to *Localized monitoring*. A good heuristic reduces the number of transitions; thus, it reduces overhead.

D. Threats to validity

Our experiments show the benefits of using adaptive monitoring instead of state-of-the-art monitoring approaches. As in every experimental protocol, our evaluation has some bias which we have tried to mitigate. All our experiments are based around the same case study. We have tried to mitigate this issue by using an available real case study. We have also used different settings across our experiments, even if all of the experiments are based on the same case study. Thus, our experiments limit the validity of the approach to applications with the same characteristics of the presented case study. New experiments with other use cases are needed to broaden the validation scope of our approach.

The evaluation of the heuristic mainly shows the potential impact of using an ideal heuristic. More case study and experiments are needed to fully validate the value of our Models@run.time based heuristic.

V. Related work

The Scapegoat framework is related to component monitoring, Models@run.time, component isolation and component

TABLE I. Features of use cases.

Test Name	Monitored Resource	Faulty Resource	Heuristic	External Task
UC1	CPU, Memory	CPU	number of failures	Weka, training neural network
UC2	CPU, Memory	CPU	number of failures	dacapo, antlr
UC3	CPU, Memory	CPU	number of failures	dacapo, chart
UC4	CPU	CPU	number of failures	dacapo, xalan
UC5	CPU, Memory	CPU	less number of failures	dacapo, chart
UC6	Memory	CPU	number of failures	Weka, training neural network

hal-00983045, version 1 - 24 Apr 2014

performance prediction approaches.

Performance and resource-consumption prediction approaches are complementary to the Scapegoat framework because they can assist in better specifying the component contracts. Some approaches require developers to provide extensive per-component metadata at design-time in order to calculate the application's overall performance or resource consumption [4], [22]. Prediction approaches have been achieved by using combinations of design-time and runtime analyses [2]. However, although many approaches to performance prediction have been proposed, none of them have obtained widespread use [23].

KAMI [19] builds performance models at design-time but uses and continually refines them at runtime. By collecting runtime data, they are able to build performance and resource consumption models that reflect real usage. They are able to adapt the application according to changes in components' behavior, but they do not use nor propose an adaptive monitoring system to minimize overhead.

State of the art monitoring systems [17], [24], [7] extract steady data-flows of system parameters, such as, the time spent executing a component, the amount of I/O and memory used, and the number of calls to a component. The overhead that these monitoring systems introduce into applications is high, which makes it unlikely for them to be used in production systems. Maurel et al. [27] propose an adaptive monitoring framework for the OSGi platform. Similarly to our work, they propose a global monitoring system that changes to a localized monitoring system when a problem is detected. However, their work is focused on CPU usage and does not consider other resources, such as, memory or I/O.

Gama and Donsez [18] propose using virtual machines in separate processes or using MVM isolates [13] to manage trusted and untrusted components. After an evaluation period, untrusted components can be moved to the trusted JVM if no problems are detected. This allows the main application to depend on potentially faulty components without risking severe crashes. We can also cite Microsoft technologies such as COM (Component Object Model) components which can be either loaded in the client application process or provided in an isolated process [26]. In addition to process virtualization, some operating systems also propose user-space virtualization, which isolates not only the processes but also the memory, the network interface and the file system. Examples of these approaches are Jails[7] for BSD, LXC[8] and CGroups for Linux, and lmctfy[9]. All of these approaches have the drawback of limiting code and instance sharing and introduce additional overhead in cross-boundary component interactions. Furthermore, depending on the complexity of the approach, there is also overhead in having to manage multiple processes.

VI. Conclusion

In this paper we presented ScapeGoat, an adaptive monitoring framework to perform lightweight yet efficient monitoring of Component-Based Systems. In ScapeGoat, each component is augmented with a contract that specifies its resource usage, such as peak CPU and memory consumption. ScapeGoat uses a global monitoring mode that has a very small overhead on the system, and a fine-grained localized monitoring mode that performs extensive checking of the components' contracts. The system switches from the global monitoring mode to the localized monitoring mode whenever a problem is detected at the global level in order to identify the faulty component. Furthermore, we proposed a heuristic that leverages information produced by the Models@run.time approach to quickly predict the faulty components.

ScapeGoat has been implemented on top of the Kevoree component framework which uses the Models@run.time approach to tame the complexity of distributed dynamic adaptations. The evaluation of ScapeGoat shows that the monitoring system's overhead is reduced by up to 80% in comparison with state-of-the-art full monitoring systems. The evaluation also presents the benefits of using a heuristic to predict the faulty component. This paper contributes to the state of the art by providing a monitoring framework which adapts its overhead depending on current execution conditions and leverages the architectural information provided by Kevoree to drive the search for the faulty component.

The work presented in this paper opens various research perspectives. Scapegoat currently uses code injection at load-time to perform fine-grain monitoring. The adaptive monitoring approach we have presented provides good results, but we believe we can reduce the overhead of CPU and memory monitoring by using a modified JVM and injecting specialized bytecode to cooperate with it. The modified JVM would account for the resources at a low-level, while the instrumentation code could provide application-level information like the component boundaries. This should result in a more efficient solution than calculating resource usage at the application-level only. A second research perspective consists in proposing an appropriate reaction when the source of a problem is discovered by ScapeGoat. Indeed, the reconfiguration when a resource-consumption problem is found could range from resource limitations for guilty components, to a replacement of the component or of part of the application. In the context of a distributed system, the set of possible reconfigurations is larger and can include moving components across the distributed infrastructure. In this context, choosing a reconfiguration to efficiently deal with the discovered fault will be necessary.

[7] http://www.freebsd.org/doc/handbook/jails.html

[8] http://lxc.sourceforge.net/

[9] https://github.com/google/lmctfy

Acknowledgement

This work has been supported by the European FP7 Marie Curie Initial Training Network RELATE (Grant Agreement No. 264840). It has been also supported by the ITEA2 MERgE project, and by the French project InfraJVM (Grant Agreement No. ANR-11-INFR-0008).

References

[1] L. Alfaro and T. Henzinger, "Interface theories for component-based design," in *Embedded Software*, ser. Lecture Notes in Computer Science, T. Henzinger and C. Kirsch, Eds. Springer Berlin Heidelberg, 2001, vol. 2211, pp. 148–165.

[2] M. Autili, P. Di Benedetto, and P. Inverardi, "A hybrid approach for resource-based comparison of adaptable java applications," *Science of Computer Programming*, 2012.

hal-00983045, version 1 - 24 Apr 2014

[3] L. Baresi, E. Di Nitto, and C. Ghezzi, "Toward open-world software: Issue and challenges," *Computer*, vol. 39, no. 10, pp. 36–43, Oct. 2006.

[4] S. Becker, H. Koziolek, and R. Reussner, "Model-based performance prediction with the palladio component model," in *Proceedings of the 6th international workshop on Software and performance*, ser. WOSP '07. New York, NY, USA: ACM, 2007, pp. 54–65.

[5] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making components contract aware," *Computer*, vol. 32, no. 7, pp. 38–45, Jul. 1999.

[6] W. Binder, "Portable and accurate sampling profiling for java," *Softw. Pract. Exper.*, vol. 36, no. 6, pp. 615–650, May 2006.

[7] W. Binder and J. Hulaas, "Exact and portable profiling for the {JVM} using bytecode instruction counting," *Electronic Notes in Theoretical Computer Science*, vol. 164, no. 3, pp. 45 – 64, 2006, proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006).

[8] W. Binder, J. Hulaas, P. Moret, and A. Villazón, "Platform-independent profiling in a virtual execution environment," *Softw. Pract. Exper.*, vol. 39, no. 1, pp. 47–79, Jan. 2009.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[10] G. S. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani, "The FRACTAL Component Model and its Support in Java," *Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[12] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama, "A component model for building systems software," in *In Proc. IASTED Software Engineering and Applications (SEA'04*, 2004.

[13] G. Czajkowski and L. Daynàs, "Multitasking without compromise: a virtual machine evolution," *ACM SIGPLAN Notices*, vol. 47, no. 4a, pp. 60–73, 2012.

[14] G. Czajkowski and T. von Eicken, "JRes: a resource accounting interface for java," in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '98. New York, NY, USA: ACM, 1998, pp. 21–35.

[15] E. M. Dashofy, A. van der Hoek, and R. N. Taylor, "An infrastructure for the rapid development of XML-based architecture description languages," in *24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 266–276.

[16] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "A dynamic component model for cyber physical systems," in *CBSE*, V. Grassi, R. Mirandola, N. Medvidovic, and M. Larsson, Eds. ACM, 2012, pp. 135–144.

[17] S. Frénot and D. Stefan, "Open-service-platform instrumentation: Jmx management over osgi," in *Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, ser. UbiMob '04. New York, NY, USA: ACM, 2004, pp. 199–202.

[18] K. Gama and D. Donsez, "A self-healing component sandbox for untrustworthy third party code execution," in *Proceedings of the 13th international conference on Component-Based Software Engineering*, ser. CBSE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 130–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13238-4_8

[19] C. Ghezzi and G. Tamburrelli, "Predicting performance properties for open systems with kami," in *Architectures for Adaptive Software Systems*, ser. Lecture Notes in Computer Science, R. Mirandola, I. Gorton, and C. Hofmeister, Eds. Springer Berlin Heidelberg, 2009, vol. 5581, pp. 70–85.

[20] V. Grassi, R. Mirandola, N. Medvidovic, and M. Larsson, Eds., *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, part of Comparch '12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012*. ACM, 2012.

[21] R. Johnson and B. Woolf, "The Type Object Pattern," 1997.

[22] M. D. Jonge, J. Muskens, and M. Chaudron, "Scenario-based prediction of run-time resource consumption in component-based software systems," in *In Proceedings of the 6th ICSE Workshop on Component-based Software Engineering (CBSE6*. IEEE, 2003, p. pages.

[23] H. Koziolek and J. Happe, "A qos driven development process model for component-based software systems," in *Proceedings of the 9th international conference on Component-Based Software Engineering*, ser. CBSE'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 336–343.

[24] H. Kreger, W. Harold, and L. Williamson, *Java and JMX: Building Manageable Systems*. Boston, MA: Addison-Wesley, 2003.

[25] M. Léger, T. Ledoux, and T. Coupaye, "Reliable dynamic reconfigurations in a reflective component model," *Component-Based Software Engineering*, pp. 74–92, 2010.

[26] J. Löwy, *COM and. NET: Component Services*. O'Reilly Media, Inc., 2001.

[27] Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi, "Adaptive monitoring of end-user osgi-based home boxes," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, ser. CBSE '12. New York, NY, USA: ACM, 2012, pp. 157–166.

[28] N. Medvidovic and R. N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Softw. Eng.*, vol. 26, pp. 70–93, January 2000.

[29] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992.

[30] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel, "Taming Dynamically Adaptive Systems with Models and Aspects," in *ICSE'09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[31] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," *Autom. Softw. Eng.*, vol. 15, no. 3-4, pp. 313–341, 2008.

[32] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani, "A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures," *Software: Practice and Experience*, 2011.

[33] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*. IEEE, 2006, pp. 65–73.

[34] The OSGi Alliance, "OSGi Service Platform Core Specification, Release 5.0," Jun. 2012, http://www.osgi.org/Specifications/.

[35] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

Cinquième partie

CV détaillé

Habilitation à Diriger des Recherches

Compte Rendu d'Activité Travaux et Titres

Olivier Barais
26/01/1980, 34 ans
Grade : 3014 - MCF CN
Université de Rennes 1
Section 27

http://olivier.barais.fr

années universitaires 2002/2014

1 **Présentation générale** . . . 1
État civil . . . 1
Expérience professionnelle . . . 2
Formation . . . 2
Informations complémentaires . . . 2
2 **Profession de foi*** . . . 3
3 **Activités d'enseignement** . . . 4
4 **Activités de recherche** . . . 8
5 **Activités d'encadrement** . . . 23
Co-encadrement de thèse . . . 23
Encadrement de stages de Master Recherche . . . 24
Autres stagiaires . . . 24
6 **Rayonnement scientifique** . . . 25
Jury de thèse . . . 25
Collaborations pédagogiques . . . 25
Comités de programmes . . . 25
Activité de relecture . . . 25
Développement logiciel . . . 26
7 **Animation scientifique** . . . 27
Participation à l'organisation de conférences et de manifestation . . . 27
8 **Ouverture sur le monde industriel** . . . 28
9 **Publications et production scientifique** . . . 30
Récapitulatif des publications pour les années universitaires 2002/2014 . . . 36
10 **Mobilité thématique et/ou géographique** . . . 37

1 Présentation générale

Olivier Barais
http://olivier.barais.fr

Doctorat d'Informatique de l'université de Lille 1
(Laboratoire d'Informatique Fondamentale de Lille et INRIA Futurs – Lille)

DEA
(Laboratoire d'Informatique Fondamentale de Lille et
Département GIP, École des Mines – Douai)

Diplôme d'ingénieur
(École des Mines – Douai)

État civil

Nom :	Olivier Barais
Date de naissance :	26 janvier 1980 à Lille (Nord), 34 ans
Nationalité :	Français
Situation familiale :	Marié, trois enfants
Adresse personnelle :	14 allée des acacias 35235 Thorignee Fouillard France
Téléphone :	(+33) 2 23 27 17 81
Adresse professionnelle :	IRISA Projet Triskell campus de Beaulieu 35 042 Rennes Cedex FRANCE
Téléphone :	(+33) 2 99 84 25 41
Fax :	(+33) 2 99 84 71 71
Mél :	barais@irisa.fr
Web :	http://olivier.barais.fr

Expérience professionnelle

09/2006 –	Maître de Conférence en Informatique à l'Université de Rennes 1 IRISA Rennes, France Lauréat de la Prime d'Excellence Scientifique depuis 2009 (2009 et 2013)
12/2005 – 08/2006	Post-doctorat INRIA au sein du projet Triskell IRISA Rennes, France Sujet : Tissage de préoccupations en phase de conception
2002 – 2005	Doctorant au sein du projet INRIA Jacquard - Équipe GOAL, Laboratoire d'Informatique Fondamentale de Lille, France Boursier MERT, moniteur de l'enseignement supérieur Sujet : Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants
2001 – 2002	Stagiaire dans le département GIP de l'école des Mines de Douai Projet de recherche : Approche dynamique de l'architecture logicielle
05/2001 – 09/2001	Stagiaire au centre de recherche et développement de Siemens AG, France Développement d'une pile d'appel H323 pour le centre d'appel VoxPortal
06/2000 – 09/2000	Stagiaire Deutschen Zentrum fur Luft und Raumfahrt (DLR), Lampoldshausen, Allemagne Développement d'une application de suivi des incidents du moteur Vulcain II

Formation

2002 – 2005	**Doctorat d'Informatique**, soutenance le 29 Novembre 2005 Université des Sciences et Technologies de Lille – Équipe GOAL au LIFL, Projet INRIA Jacquard Mention Très Honorable Sujet : Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants
2001 – 2002	**DEA d'informatique fondamentale**, Mention Bien Université des Sciences et Technologies de Lille, France Sujet : Approche statique, dynamique et globale de l'architecture d'applications réparties
1998 – 2002	**Diplôme d'ingénieur** Ecole des Mines de Douai, France Spécialité : ingénierie des sciences de l'information et de la communication
1997 – 1998	**Classe préparatoire à l'entrée aux grandes écoles** Lycée Henri Wallon (Valenciennes), académie de Lille Série : Math, Physique et Sciences de l'ingénieur
Juin 1997	**Baccalauréat scientifique**, Mention Très Bien Lycée Sainte Odile (Lambersart), académie de Lille

Informations complémentaires

Activités extra-professionnelles :

- Sports : volley-ball
- Centres d'intérêt : lecture, sports nautiques

2 Profession de foi*

Enseignant-chercheur depuis 8 ans, j'essaie de mener de manière équilibrée les différentes facettes de mon métier en m'investissant à la fois dans l'administration des filières d'enseignement, tout en conservant un investissement important dans les activités naturelles à la base de notre statut :

enseignement, mise en œuvre de TP, création de projets, ...

recherche, réalisation de prototype, rédaction d'articles et encadrement, ...

transfert, montage de projets collaboratifs et transfert vers le milieu industriel.

Les tâches à caractère administratif, les activités d'animation et celles d'expertise font partie intégrante du métier d'enseignant-chercheur et les responsabilités que j'ai prises à l'Université de Rennes 1 (Responsable du Master 2 Génie Logiciel, Coordinateur des Master 2 professionnel en informatique de l'université de Rennes 1 (5 spécialités), membre de l'équipe de direction de l'UFR témoignent de ma volonté de m'insérer totalement dans le fonctionnement de l'Université et de travailler en équipe pour construire de nouveaux projets.

Il est pour moi évident que le travail d'équipe prime et si j'arrive à mener conjointement l'ensemble des facettes de mon métier avec plaisir c'est que je ne travaille pas seul. Nombreux sont ceux qui peuvent en témoigner et à qui je dois beaucoup :

- Ma directrice de thèse, Laurence Duchien, qui m'a mis le pied à l'étrier du monde de la recherche et qui reste toujours disponible et de très bon conseil.
- Jean-Marc Jézéquel qui m'a accompagné très fortement en début de carrière et avec qui je garde un plaisir immense à travailler tant d'un point de vue enseignement, recherche ou dans les missions plus administratives de l'université.
- Les membres de mon équipe de recherche *Triskell* maintenant *DiverSE* avec qui j'ai la chance de travailler quotidiennement dans une ambiance de travail toujours excellente. A leur contact, j'ai progressé et appris tous les jours. La qualité de l'esprit d'équipe qu'a su insuffler Jean-Marc Jézéquel et qu'a su entretenir Benoit Baudry par la suite m'ont toujours beaucoup apporté.
- Les doctorants avec qui j'ai eu la chance de mener des aventures scientifiques, Brice, Gégory, Mickaël, Erwan, François, Inti, Emmanuelle, Paul, Bosco, Julien, Mohammed, Thomas. J'ai appris énormément à leur contact et je garde un vrai plaisir à échanger avec eux.
- Jérôme Le Noir et son équipe de Thales Research & Technology avec qui j'ai eu la chance de collaborer depuis 2008 dans des conditions chaleureuses fondées sur une confiance réciproque.
- Mes collègues de l'UFR ISTIC ou de l'ESIR. L'organisation des formations, les réunions pédagogiques, la préparation des habilitations de formation ont été et seront toujours des moments passionnants pour lesquels ce métier reste assez unique.
- Mes anciens collègues lillois que j'ai toujours grand plaisir à retrouver et avec qui j'échange régulièrement.
- Mes étudiants de Master avec qui le contact a toujours été plaisant. Ils m'ont apporté beaucoup par leur curiosité et leur questionnement.

*(Largement inspiré de textes de plusieurs collègues avec qui je partage la même vision de notre métier)

3 Activités d'enseignement

Mes enseignements peuvent être regroupés en trois thèmes principaux :

1. génie logiciel et architecture logicielle,
2. les nouvelles technologies,
3. la gestion de projet.

Dans l'ensemble de ces domaines, je cherche à avoir une approche du *bottum-up* en cherchant à permettre aux étudiants de trouver les abstractions fournies par certaines nouvelles technologies, à leur faire comprendre par l'exemple les concepts inhérents au développement logiciel moderne et à appréhender par l'exemple les problématiques de la gestion de projet. Mon activité d'enseignement parfois lourde en terme d'horaire a été rendue possible par la participation à une équipe pédagogique menée par Jean-Marc Jézéquel. Cette équipe pédagogique permet une confrontation des méthodes de travail et une mise en commun des ressources. En outre, au sein de ce volume horaire important se sont toujours greffés des suivis de projet et des suivis de stage formateurs pour les étudiants, mais aussi pour moi-même. Ces suivis de stages et de projets me permettent d'assurer une activité de veille technologique et d'apprendre de nouveaux domaines fonctionnels et de nouvelles technologies. Durant ces 8 dernières années, je suis intervenu dans différentes filières de Master (cours, travaux dirigés, travaux pratiques et encadrement de stagiaires. Les trois sections qui suivent décrivent respectivement les formations où je suis intervenu, les cours que j'ai assurés et enfin les responsabilités que j'ai assurées au sein de l'Ifsic puis de l'ISTIC. En outre, j'ai eu la chance d'intervenir aussi à l'INSA, l'ESIR, l'ENSAI, Telecom Bretagne, Agrocampus. Enfin, dans le cadre de la formation professionnelle, je suis intervenu à la DGA ou au sein de différentes entreprises.

3.1 Formations

- Master d'Informatique (première année)
- Master d'Informatique (deuxième année) (spécilité GL, IR et SSI)
- Master Miage (première année)
- Master Miage (deuxième année)
- DU Génie Logiciel (Formation continue / Bac +4).
- DIIC, Diplôme d'Ingénieur en Informatique et téléCommunications de l'Ifsic.
  - DIIC3, 3e année du DIIC (Bac +5).
- ESIR, Ecole Supérieur d'Ingénieur de Rennes (Université de Rennes 1).
  - ESIR 2, 2e année du DIIC (Bac +4).
- ENSAI Ecole Nationale de la Statistique et de l'Analyse de l'Information.
  - 3 ème année filière SIS (Système d'Information Statistique)
- INSA Rennes
  - 2ème année

3.2 Modules enseignés

ABC (Architecture à Base de Composants) en M2 GL et M2 Miage
Volume horaire : 14h CM et 18h TP - de 2006 à 2008

TAA (Technique Architecture Logicielle Avancée) en M2 GL
Volume horaire : 14h CM et 18h TP depuis 2008

GLA (Génie Logiciel Appliqué) en M2 IR et M2 SSI
Volume horaire : 14h CM et 18h TP depuis 2008

SIO (Système d'Information Objet) en M1 Miage
Volume horaire : 8h CM, 8h TD et 12h TP de 2006 à 2012

SIR (Système d'Information Réparti) en M1 Miage
Volume horaire : 12h CM, 12h TD et 14h TP depuis 2012

MDO (Modèlisation et Design par Objet) en M2 GL
Volume horaire : 8h TD et 8h TP de 2006 à 2008

V&V (Vérification et Validation) en M2 GL
Volume horaire : 16h TP de 2007 à 2010

ACF (Analyse et Conception Formelle) en M1 Informatique
Volume horaire : 20h TP de 2011 à 2012

CAO (Conception et Analyse par Objet) en M2 GL
Volume horaire : 4h TD et 12h TP de 2006 à 2010

IM (Ingénierie des Modèles) en DIIC3
Volume horaire : 6h TD et 6h TP de 2006 à 2009

IDM (Ingénierie Dirigé par les Modèles) à l'Insa
Volume horaire : 6h CM et 6h TP de 2007 à 2009

ADT (Analyse, Développement et Tests) en M1 Informatique
Volume horaire : 20h TD et 14h TP de 2006 à 2008

ACO (Analyse et Conception par Objets) en M1 Informatique
Volume horaire : 20h TD et 14h TP de 2008 à 2010

GPL (Gestion de Projet Logiciel) en M1 Miage
Volume horaire : 12h TD et 12h TP de 2006 à 2012

Projet en Entreprise en M1 Miage
Volume horaire : 32h TD depuis 2006

Projet en ESIR 2ème année
Volume horaire : 34h TD depuis 2013

UML (Mise à niveau en UML) en M2 ISTIC
Volume horaire : 4h CM et 4h TD de 2006 à 2012

UML (Le langage UML) en DUGL
Volume horaire : 8h CM et 8h TD en 2007

MoR (Mise à niveau objet relationnel) en M2 ISTIC
Volume horaire : 4h CM et 4h TP depuis 2008

GL (Génie Logiciel) à l'ENSAI
Volume horaire : 20h CM depuis 2008

Outre ces modules, je suis intervenu ponctuellement pour des formations professionnelles pour la DGA ou des entreprises comme BenchMark Group ou EDS. Je suis aussi intervenu ponctuellement à AgroCampus ou Telecom Bretagne pour des cours de conception objets.

3.3 Responsabilités et autres activités pédagogiques

En plus des responsabilités inhérentes au déroulement des cours tels que la conception de séries d'exercices et de sujets travaux pratiques, la correction des TP, la rédaction de sujets d'examen, la surveillance des

épreuves et la correction des copies, j'ai assuré d'autres responsabilités parmi lesquelles : responsabilité de diplôme, équipe de direction de l'UFR, membre de jurys de diplôme.

3.3.1 Responsabilité de diplôme

De septembre 2009 à septembre 2013, j'ai été responsable du Master 2 spécialité Génie Logiciel de l'ISTIC. Cette spécialité accueille sur une année (bac +5) autour de 35 étudiants par promotion. La responsabilité de la filière implique entre autres, le recrutement en accès direct d'étudiants, la disponibilité par rapport aux doléances des étudiants et aux problèmes que certains peuvent rencontrer, l'assurance du bon déroulement des cours, l'évolution des programmes, la recherche d'intervenants extérieurs, l'organisation des soutenances de stage et bien sûr la présidence des jurys et la validation des stages de fin d'études en entreprise.

De plus de septembre 2009 à septembre 2013, j'ai coordonné les cinq spécialités de Master de la mention informatique de l'université de Rennes 1.

3.3.2 Équipe de direction de l'UFR

Depuis septembre 2009, je suis membre de l'équipe de direction de l'UFR d'informatique et d'électronique de l'université de Rennes 1.

3.3.3 Jurys

J'ai aidé ou je suis membre de nombreux jurys de diplômes au sein de l'UFR

- M1 Miage,
- M2 Miage,
- M2 GL,
- M2 MITIC,
- M2 ITEA

3.3.4 Habilitation de diplôme

J'ai participé aux campagnes d'habilitation des diplômes de Master de l'ISTIC en 2008 et 2012. En 2008, j'ai participé sous la tutelle d'Isabelle Puaut à la réflexion autour des contenus de formation du Master en informatique de l'IFSIC pour l'habilitation 2008-2012. J'ai sur ce point animé les discussions autour des contenus du futur axe génie logiciel du Master 1 en informatique. Pour 2012, en tant que responsable de Master, j'ai piloté avec Isabelle Puaut, Sandrine Blazy et Thomas Genet la rédaction de ce document d'habilitation.

3.3.5 Journées pédagogiques

J'ai organisé les journées pédagogiques de l'Ifsic à Saint Malo en 2007. En juillet 2007, comme second de Jean-Marc Jézéquel sur le thème de l'ingénierie dirigée par les modèles. Les journées pédagogiques étaient des rencontres informelles entre enseignants de l'Ifsic principalement qui étaient organisées annuellement et dont le but est de discuter de l'enseignement d'une matière ou de la pédagogie d'un enseignement. Au cours de ces journées (3 jours), il est organisé des séminaires et des séances de travaux pratiques où les enseignants repassent de l'autre côté du pupitre. Chaque année voit la participation d'un trentaine d'enseignants de l'Ifsic et d'une dizaine de locaux (enseignants de l'IUT Vannes ou de l'IUT de Lannion).

3.4 Prime d'Excellence Scientifique

Je suis titulaire de la prime d'excellence scientifique depuis septembre 2009 (sessions 2009 et 2013).

3.5 Comité de sélection

Depuis 2009, j'ai été membre de plusieurs comités de sélection de Rennes 1 (*7), des universités de Brest, Bordeaux (*2) et Nantes (*2).

4 Activités de recherche

Je mène mon activité de recherche au sein de l'Irisa (Institut de recherche en informatique et systèmes aléatoires) qui est une unité mixte de recherche (UMR 6074) en informatique, en traitement du signal et des images, et en robotique. Membre d'une équipe INRIA, je suis associé à INRIA au sein du centre Inria Rennes-Bretagne Atlantique. De 2002 à décembre 2005, j'ai été membre de l'équipe Jacquard sous la direction de Jean-Marc Geib puis de Laurence Duchien. Depuis décembre 2005 (début de mon post-doctorat) j'ai été membre de l'équipe Triskell (Model-Driven Engineering for Component-Based Software) dirigée par Jean-Marc Jézéquel jusqu'en 2011 et par Benoit Baudry depuis janvier 2012. Depuis septembre 2013, je suis détaché à 20 % au sein de l'IRT BCOM. Depuis janvier 2014 je suis membre de l'équipe-projet INRIA DiverSE dirigé par Benoit Baudry.

Cette partie de mon CV propose tout d'abord un résumé court de mon doctorat et de ma proposition de synthèse en vue de l'habilitation à diriger des recherches. Il présente ensuite le contexte dans lequel j'effectue ma recherche, synthétise les principales contributions obtenues au travers du co-encadrement de plusieurs thèses. Enfin, je présente les questions de recherche et les perspectives de recherche que je souhaite aborder dans le futur.

Doctorat en Informatique de l'université de Lille 1

Laboratoire de recherche : LIFL
Équipe d'accueil : Équipe GOAL, Projet INRIA Jacquard
Directeur de thèse : Laurence Duchien, Professeur à l'Université de Lille 1
Financement : MERT + monitorat CIES
Date de début : 1er octobre 2002
Date de soutenance : 29 Novembre 2005

Jury :

Présidente	Sophie Tison, Professeur à l'Université de Lille I
Rapporteurs	Jacky Estublier, Directeur de Recherche au Centre National de la Recherche Scientifique (Grenoble)
	Jean-Marc Jezequel, Professeur à l'Université de Rennes I
Examinateurs	Thomas Ledoux, Enseignant-chercheur à l'École des Mines de Nantes
	Tom Mens, Professeur à l'Université de Mons-Hainaut (Belgique)

Titre : Construire et Maîtriser l'Évolution d'une Architecture Logicielle à base de Composants

Résumé des travaux

Les travaux présentés dans ma thèse se situent dans le domaine de l'architecture logicielle. Ma thèse a proposé un cadre de description d'architectures logicielles complet, permettant de spécifier une architecture logicielle de manière incrémentale en travaillant sur l'analyse d'une description d'architecture et son utilisation pour le prototypage rapide d'une application. Deux axes principaux ont structuré ces travaux. Le premier axe a permis la définition d'un modèle abstrait de description d'architecture logicielle à base de composants. Ce modèle rend possible la validation d'une description d'architecture logicielle et la génération de code vers différentes plates-formes à composants pour un prototypage rapide de l'application. Le deuxième axe étend ce modèle afin de favoriser l'intégration de nouvelles préoccupations au sein d'une architecture logicielle. Inscrit dans une démarche de séparation des préoccupations au niveau de la conception de l'application, j'ai proposé dans ma thèse un mécanisme de transformation permettant de tisser de façon fiable ces nouvelles préoccupations au sein d'une architecture logicielle existante.

Résumé de la proposition d'habilitation à diriger des recherches

Le développement logiciel « *traditionnel* », généralement fondé sur l'hypothèse d'un monde clos définissant une frontière connue et stable entre le système et son environnement n'est plus tenable. Par opposition, la notion de système dit ouvert et éternel s'est imposée à la plupart des systèmes informatiques. Ces systèmes logiciels se caractérisent par leur besoin d'offrir des capacités d'adaptation qui leur permettent de réagir aux changements de leur environnement de manière continue et sans interruption de service.

Un des challenges important pour la communauté du génie logiciel est d'identifier et de supprimer progressivement les limites liées à l'hypothèse du monde clos. En partant de cette hypothèse dit de monde ouvert, cette habilitation expose les bénéfices engendrés par l'effacement de la frontière entre la phase de conception et la phase d'exécution du logiciel en proposant l'utilisation des travaux liés à la modélisation non plus uniquement lors de la phase de conception du système, mais aussi au cours de l'exécution des systèmes dits ouverts.

Pour ce faire, cette habilitation synthétise dans un premier temps les fondations d'une approche permettant l'utilisation de techniques de modélisation, à l'exécution en se concentrant principalement sur le point de vue de l'architecte logiciel. Nous exposons ensuite les bénéfices attendus en montrant comment des approches avancées de composition logicielle, de vérification ou de gestion de la variabilité peuvent être bénéfiques pour la compréhension et la maîtrise de l'espace de configuration et de reconfiguration d'un système dit ouvert. Nous synthétisons ensuite les principaux challenges liés à l'utilisation de techniques de modélisation à l'exécution en particulier dans le cadre de systèmes distribués et hétérogènes. Enfin, nous validons cette approche en la confrontant à différents domaines d'applications : le monde des objets connectés, le *cloud computing* et le web. Pour chacun de ces domaines, nous cherchons à vérifier le domaine de validité de cette idée d'utilisation de modélisation à l'exécution en regardant sa pertinence pour chaque domaine étudié par rapport à ses propres contraintes.

4.1 Contexte de ma recherche

La conception de logiciels de grande taille est aujourd'hui l'un des enjeux majeurs en informatique. Citons pour cela trois trivialités classiques, mais néanmoins clés et de plus en plus vraies :

— la complexité de ces systèmes croît. Prenons par exemple les applications de commerce électronique, les systèmes d'information d'entreprise, les systèmes d'exploitation, les services web, etc. Ces applications tendent à fournir toujours plus de fonctionnalités, contribuant ainsi à accroître d'autant leur taille et leur complexité. L'utopie d'une convergence des plateformes ou des protocoles n'a pas tenu ses promesses et la problématique de la gestion de la distribution et de l'hétérogénéité des plateformes reste une problématique importante dans la construction de tels systèmes.

— la dépendance de nos sociétés vis-à-vis des systèmes informatiques a atteint un point de non-retour que conforte le caractère de plus en plus invisible et de plus en plus enfoui de l'immense majorité de ces systèmes.

— Le développement « *traditionnel* »de logiciels, généralement fondé sur l'hypothèse du monde clos, définissant une frontière connue et stable entre le système et son environnement n'est plus tenable. La notion de système dit ouvert et éternel, dont le logiciel a besoin d'offrir des capacités d'adaptation, qui lui permettent de réagir aux changements de son environnement de manière continue et sans interruption de service, est une réalité.

L'ensemble de mes recherches s'est placé dans ce contexte de systèmes ouverts. Trois approches, constats ou tendances ont fortement influencé ce travail de recherche.

La caractérisation des systèmes dynamiquement adaptable DAS) Une première approche prometteuse consiste à concevoir et à implémenter ces systèmes dits ouverts comme des systèmes adaptatifs (*DAS, Dynamically Adaptive Systems*), qui peuvent s'adapter selon leurs contextes d'exécution, et évoluer selon les exigences utilisateur. Il y a plus d'une décennie, Peyman Oreizy et al. [OGT+99b] ont défini l'évolution comme "l'application cohérente de changements au cours du temps", et l'adaptation comme "le cycle de monitoring du contexte, de planification et de déploiement en réponse aux changements de contexte". Cette frontière entre évolution logicielle et adaptation dynamique a tendance à disparaître dans les systèmes adaptatifs.

Les systèmes adaptatifs ont des natures très différentes :

— systèmes embarqués [HGC+06] ou systèmes de systèmes [BS06, Got08],

— systèmes purement auto-adaptatifs ou systèmes dont l'adaptation est choisie par un être humain

Si l'on cherche à modéliser la logique d'adaptation et quelque soit son type, l'exécution d'un DAS peut être abstrait comme une machine à états [BBFS08, ZC06], où :

— **Les états** représentent les différentes configurations (ou les modes) possibles du système adaptatif. Une configuration peut être vue comme programme "normal", qui fournit des services, manipule des données, exécute des algorithmes, etc.

— **Les transitions** représentent toutes les différentes migrations possibles d'une configuration vers une autre. Ces transitions sont associées à des prédicats sur le contexte et/ou des préférences utilisateur, qui indiquent quand le système adaptatif doit migrer d'une configuration à une autre.

Fondamentalement, cette machine à états décrit le cycle d'adaptation du DAS. L'évolution d'un DAS consiste conceptuellement à mettre à jour cette machine à états, en ajoutant et/ou enlevant des états (configurations) et/ou des transitions (reconfigurations).

L'énumération de toutes les configurations possibles et des chemins de migration reste possible pour de petits systèmes adaptatifs. Ce design en extension permet de simuler et de valider toutes les configurations et reconfigurations possibles, au moment du design [ZC06], et de générer l'intégralité du code lié à la logique d'adaptation du DAS.

Cependant, dans le cas de systèmes adaptatifs plus complexes, le nombre d'états et de transitions à spécifier explose rapidement [MBJ+09, FS09] : le nombre des configurations explose d'une manière combinatoire par rapport aux nombres de *features* dynamiques que le système propose, et le nombre de transitions est quadratique par rapport au nombre de configurations. Concevoir et mettre en application la machine à

états dirigeant l'exécution d'un système adaptatif complexe (avec des millions ou même des milliards de configurations possibles) est une tache difficile et particulièrement propice aux erreurs.

L'architecture logicielle à base de composants Une deuxième approche complémentaire à la notion de DAS pour maîtriser la complexité et offrir un support de la variabilité à l'exécution est issue du domaine de l'architecture logicielle. *L'architecture logicielle* fournit une abstraction et une approche modulaire dans la conception de système en explicitant la notion de composant logiciel, de connecteur et de configuration. L'idée de composant logiciel date déjà d'un certain nombre d'années. On trouve une des premières utilisations explicites de ce terme en 1968 [McI68]. Cependant, il a fallu attendre la fin des années 90 et la mise en évidence des lacunes du paradigme objet dans le domaine de la réutilisation pour voir apparaître un regain d'intérêt autour de la notion de composant logiciel comme unité de réutilisation de modules logiciels. Pour contrer le problème inhérent à la programmation par objets, identifié sous le nom de *Fragile Base Class* [Szy96, MS98], l'approche par composants propose une meilleure séparation entre la partie réalisation d'un module logiciel et son interface, c'est-à-dire la description de services qu'il fournit et qu'il requiert. Chaque composant est une boîte noire, indivisible, composable, déployable et identifiable par les services qu'elle offre et qu'elle requiert.

Le découplage entre la partie réalisation d'un composant et son interface avec l'environnement met en avant deux informations : une description claire de l'interface des composants et une description de l'assemblage des composants, c'est-à-dire leurs interactions. Ces deux informations au cœur de l'approche par composants constituent les éléments de base de ce que l'on nomme l'*architecture logicielle*. L'utilisation d'une description d'architecture logicielle apporte quatre avantages dans un projet informatique [GS93]. Elle facilite la compréhension de la structure d'un système. Elle permet son analyse. En tant que cadre pour la construction de l'application, elle permet la génération de code et l'identification des opportunités de réutilisation. Enfin, elle définit les invariants du système et sert donc de pivot pour son évolution.

De l'agilité à la livraison de logiciel en continu. Le troisième constat qui a fortement influencé mes travaux de recherche est lié à l'évolution des méthodologies de développement qui ont également connu des modifications profondes pour répondre au besoin d'évolution continue de ces logiciels. Si le cycle en V était préconisé dans les années 80, à savoir conception et spécification initiale puis génération ou implémentation de code, les méthodes modernes préconisent l'hyper agilité[Sto09]. En effet dans une étude publiée en 2011 [Ver10] sur l'adoption industrielle des méthodes Agiles, *VersionOne* annonce que 80% des sondés déclarent que leur compagnie a adopté un tel processus. Ces nouvelles méthodes cherchent à introduire des cycles plus courts de développement afin de répondre plus rapidement à des évolutions de spécification et surtout à obtenir très tôt des retours utilisateurs. Cette hyper agilité s'accompagne d'une nouvelle approche pour le test et l'intégration des nouveaux développements : l'intégration continue. Dans cette approche un système de tests est remis à jour avec les derniers artefacts de développement en continu, le plus souvent sans ré-installation particulière de logiciel. Cette méthodologie rapproche d'autant plus ces systèmes non critiques des contraintes spécifiées pour les DAS, étayant la thèse de rapprocher les abstractions des deux mondes. Le cycle de développement et de livraison se fait donc maintenant de manière continue sur des systèmes critiques ou non.

4.2 Challenges Liés à l'Ingénierie des Systèmes Adaptifs Complexes

Betty Cheng *et al.* ont identifié plusieurs défis liés aux DAS [CLG+09], spécifiques à différentes activités : modélisation des différentes dimensions d'un DAS, expression des besoins, ingénierie (design, architecture, vérification, validation, etc) et assurance logicielle. Ces défis peuvent être complétés par l'analyse de Baresi *et al.* sur les challenges pour la construction de systèmes ouverts. La plupart de ces défis peuvent se résumer à trouver le niveau juste de l'abstraction :

- assez abstrait pour pouvoir raisonner efficacement et exécuter des activités de validation, sans devoir considérer tous les détails de la réalité,
- et suffisamment détaillé pour établir le lien (dans les 2 directions) entre l'abstraction et la réalité c.-à-d., pour établir un lien causal entre un modèle et le système en cours d'exécution.

L'abstraction est l'une des clés pour maîtriser la complexité des logiciels. La clé pour maîtriser les systèmes dynamiquement adaptatifs [MBJ+09] est de réduire le nombre d'artefacts qu'un concepteur doit spécifier

pour décrire et exécuter un tel système, et d'élever le niveau d'abstraction de ces artefacts. Selon Jeff Rothenberg [RWLN89],

> *Modéliser, au sens large, est l'utilisation rentable d'une chose au lieu d'une autre pour un certain but cognitif. Cela permet d'employer quelque chose qui est plus simple, plus sûr ou meilleur marché que la réalité, pour un but donné. Un modèle représente la réalité pour un but donné ; c'est une abstraction de la réalité dans le sens qu'il ne peut pas représenter tous les aspects de réalité. Cela permet d'envisager le monde d'une façon simplifiée, en évitant la complexité, le danger et l'irrévocabilité de la réalité.*

C'est dans cette optique et pour répondre à la complexité d'assemblage des systèmes modernes qu'a émergé la mouvance de méthodes et d'outils autour de l'ingénierie dirigée par les modèles (IDM). L'IDM appliquée à la construction de logiciel vise donc à fournir des possibilités de points de vue différents et simplifiés d'un système informatique. Ces outils ont été exploités et ont démontré leur intérêt lors de la phase de conception d'un système permettant de vérifier, d'évaluer des propriétés sur ce dernier ou de produire automatiquement une partie de celui-ci.

Dans les DAS complexes, un des aspects importants de la réalité que nous voulons abstraire est entre autres le système lui-même. Ceci soulève les questions suivantes :

1. Que proposer dans cette abstraction afin que l'architecte d'un système adaptatif distribué et hétérogène puisse spécifier, déployer, reconfigurer et exécuter un tel système ? Quels concepts permettent de capturer les aspects essentiels d'un système distribué dynamiquement adaptable en cours d'exécution ? Une telle couche d'abstraction doit expliciter et non masquer les problèmes du domaine d'étude. Ceci est nécessairement vrai dans le monde du distribué en accord avec le constat de Guerraoui et al [GF99] qui parlent de : ≪ mythe de la distribution transparente ≫mais aussi dans un monde hétérogène ou l'uniformisation par l'intersection des propriétés des constituants du système se retrouve toujours être inutile sur des cas réels.
2. Comment établir un lien entre la configuration (architecture) correspondant à un ensemble de fonctionnalités, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation ?
3. Comment faire migrer un DAS distribué de sa configuration courante vers une nouvelle configuration sans devoir écrire à la main tous les scripts de bas niveau et spécifiques à une plateforme d'exécution donnée ?
4. Comment maintenir la cohérence de ce modèle représentant l'état du système dans un environnement distribué ? Ce point reste un challenge particulièrement dans des environnements mobiles où les communications sont intermittentes ?
5. Comment valider la pertinence de l'abstraction proposée et sa pertinence pour le domaine visé ?
6. Comment réutiliser les approches, méthodes et outils habituellement utilisés sur un modèle de conception, à l'exécution. ?

Ces questions amènent à élever le niveau d'abstraction, et à réduire le nombre d'artefacts requis pour spécifier et exécuter des systèmes distribués dynamiquement adaptables, tout en préservant un degré élevé de validation, d'automatisation et d'indépendance par rapport à des choix technologiques (par exemple, outils utilisés au design, plateformes d'exécution, système de règles pour diriger l'adaptation dynamique, etc.).

Mais ces questions amènent aussi à adapter les techniques de modélisation pour la prise en compte de l'hypothèse du monde ouvert et leur utilisation à l'exécution.

Les trois sous-sections suivantes synthétisent les principaux résultats obtenus face à ces challenges.

4.3 Vers une convergence de l'espace de conception et de l'espace d'exécution

4.3.1 Contribution (i) : un modèle à l'exécution [32,37,34,44,45,48,69,73]

Daniel G. Bobrow et al. [BGW93] ont défini la réflexion comme :

> *La capacité d'un programme à manipuler comme données quelque chose représentant l'état du programme lui-même pendant sa propre exécution. Il y a deux aspects d'une telle manipulation : introspection et intercession. L'introspection est la capacité d'un programme d'observer et donc*

raisonner au sujet de son propre état. L'intercession est la capacité d'un programme à modifier son propre état d'exécution ou de changer son interprétation.

Cela consiste fondamentalement à maintenir un lien causal entre la réalité (c.-à-d. le système en cours d'exécution) et un modèle de la réalité. Un changement significatif du système en cours d'exécution met à jour le modèle, et n'importe quelle modification significative du modèle affectera le système en cours d'exécution. Cela contredit la citation de Jeff Rothenberg puisque ce modèle de réflexion n'évite pas le danger et l'irrévocabilité, en raison de la synchronisation forte entre la réalité et le modèle.

La réflexion est un concept fondamental pour mettre en œuvre les systèmes adaptatifs, mais nous proposons d'aller plus loin dans nos travaux afin d'y inclure les arguments de Jeff Rothenberg. Le modèle doit toujours refléter ce qui arrive dans le système courant (introspection), pareillement à un miroir, de sorte qu'il soit possible de toujours raisonner sur un modèle à jour. Cependant, il faut offrir plus de liberté pour manipuler, transformer, valider, etc. le modèle sans modifier directement le système en cours d'exécution. C'est le principe de base de l'intelligence humaine (et de l'intelligence artificielle [RNC+95] dans une certaine mesure). Un humain peut établir mentalement plusieurs modèles potentiels de la réalité et évaluer mentalement ces modèles au moyen de scénarios [BBF09] : que se produirait-il si je faisais cette action ? Pendant ce raisonnement mental, la manipulation du modèle n'affecte pas la réalité, tant qu'une solution acceptable n'ait été trouvée. Enfin, cette solution est effectivement mise en œuvre, ce qui impacte la réalité. En d'autres termes, le modèle mental est re-synchronisé avec la réalité. Dans le cas où un aspect de la réalité change pendant le processus de raisonnement, le modèle est mis à jour et le processus de raisonnement reconstruit des modèles mentaux, idéalement en mettant à jour les modèles déjà existants.

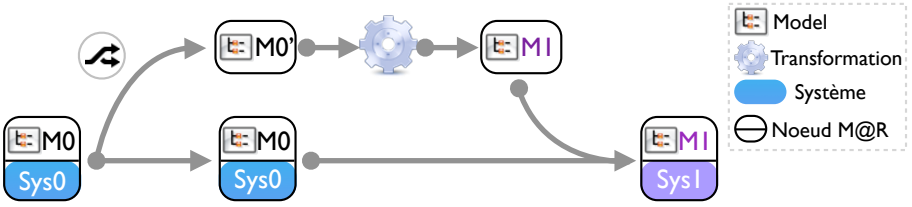
En réponse au besoin d'offrir une couche de réflexion plus facilement utilisable lors des phases de raisonnement et plus riche en termes de méta-données, nous avons proposée l'idée de rapprocher les activités de modélisation et les systèmes concrets via le paradigme nommé *"Model@Runtime"* (M@R). D'abord imaginée encore une fois comme vision contemplative du système à des fins de simulations[OGT+99a],[BBF09],[ZC06], cette approche repose sur une représentation abstraite sous forme de modèles en permanence «à jour» du das. Cette couche modèle devient alors une couche d'introspection donnant la capacité d'observer et de raisonner sur l'état du système. Cette couche d'introspection permet une analyse de l'état du système en apportant une navigation aisée dans une représentation schématique (simplifiée) des différents éléments qui le composent.

Dans cette tendance du *"Model@Runtime"*, les travaux de Brice Morin[Mor10] que j'ai la chance d'encadrer avec Jean-Marc Jézéquel ont promu cette couche modèle non plus uniquement comme couche d'introspection mais également d'intercession permettant de modifier un système au travers de sa représentation réflexive modèle. En accord avec la définition de Bobrow *et al.*[BGW93] l'intercession est la capacité d'un système à modifier son état interne. L'approche m@r de Morin pour les das vise justement la construction de système réflexif alliant une capacité d'introspection et d'intercession au travers de la même couche de modélisation. En tant que couche de modélisation abstraite, la couche de réflexion ainsi proposée est désynchronisable, permettant ainsi de faire plusieurs modifications au niveau modèle, et de les tester avant de les appliquer sur le système réel.

De manière schématique cette approche permet d'extraire et modifier le modèle réflexif du système concret (ex : ajout/suppression d'un composant), puis de détecter et déployer toute modification par différence de versions vers la plate-forme (ex : déploiement du composant). Inversement, toute modification de la plate-forme extérieure donne lieu à une modification dans la couche de m@r introduisant alors un lien causal bidirectionnel. Le M@R devient donc une couche qui peut être exploitée de manière asynchrone : en d'autres termes, il est possible de modifier celle-ci de manière indépendante de la plate-forme puis la synchroniser par la suite. Ceci permet par exemple d'introduire des étapes de vérification avant déploiement, mais surtout de ne pas contraindre les capacités de modification du m@r avec des contraintes de la plate-forme. Par exemple, si un composant A dépend d'un autre B, la plate-forme va exiger que A soit déployé avant B pour respecter les contraintes de plates-formes. A l'inverse, l'ajout de B avant A dans le modèle n'est pas soumis aux mêmes contraintes d'ordres si l'application de l'adaptation se fait après les deux ajouts. Les contraintes d'une couche réflexive liée à sa plate-forme d'exécution restreignent alors toutes les approches de génération automatique d'adaptations. A l'inverse en retardant ces contraintes au moment de l'application du modèle réflexif, les approches génératives peuvent ainsi manipuler le modèle dans n'importe quel ordre.

Cette capacité à désynchroniser le modèle réflexif et la plate-forme est illustrée par la figure 1 et permet d'exploiter au *runtime* ce modèle comme une cible pour tous les algorithmes de synthèse de modèles qui servent à calculer la meilleure architecture.

FIGURE 1 – Illustration du processus Model@Runtime désynchronisable



4.3.2 Contribution (ii) : nouvelles propositions dans le domaine de gestion de la variabilité et de la composition logicielle

Gestion de la Variabilité, Dérivation de Configurations Comme nous l'avons déjà mentionné, spécifier et mettre en oeuvre explicitement la machine à états dirigeant l'exécution d'un DAS devient rapidement difficile en raison de l'explosion combinatoire du nombre de configurations, et de l'explosion quadratique du nombre de transitions.

Tandis qu'il est encore possible de définir cette machine à états pour des petits systèmes adaptatifs [BGFB08, ZC06], cela devient presque impossible pour des systèmes adaptatifs complexes, avec des millions ou même des milliards de configurations [FS09, MBJ+09]. Même si les états et les transitions sont spécifiés à un haut niveau d'abstraction, il devient rapidement difficile pour un humain, et même pour une machine, de définir un nombre si élevé d'artefacts [FS09, MBJ+09].

Sven Hallsteinsen *et al.* ont conceptualisé les DAS comme des lignes de produits dynamiques [HSSF06, HHPS08] (DSPL, Dynamic Software Product Lines) dans lesquelles la variabilité est fixée jusqu'à l'espace d'exécution. Semblables aux lignes de produits traditionnelles [CN01] (SPLs), l'idée fondamentale est de capitaliser sur les parties communes et de contrôler précisément les variations entre les différents produits. Ainsi, le nombre élevé de configurations possible pour un DAS est décrit en intention, plutôt qu'en extension.

Dans une SPL, les produits sont la plupart du temps dérivés selon des décisions humaines, de l'expression des besoins (*requirement*) au déploiement [GBS01]. En revanche, le processus de dérivation d'une DSPL est plus complexe et plus varié. À la différence d'une SPL "classique", les produits d'une DSPL sont fortement dépendants : le système doit commuter dynamiquement et sans risque de sa configuration courante vers une autre configuration. La décision de dériver un produit (c.-à-d. s'adapter vers une autre configuration) dépend du type de système adaptatif :

- dans les systèmes Auto-Adaptatifs (*SAS, Self-Adaptive Systems*), par exemple des systèmes adaptatifs embarqués dans des avions [OGT+99b], le processus de dérivation est entièrement automatisé et suit souvent la boucle autonomique MAPE (Monitor-Analyze-Plan-Execute) [KC03].
- dans les systèmes dont l'adaptation est dirigée par un être humain, par exemple un système domotique [NDBJ08], le choix des features à intégrer est la plupart du temps dirigé par l'utilisateur final.

La communauté SPL [BGH+06, CN01, ZJ06] propose déjà des formalismes et des notations, tels que les feature diagrams, pour décrire une gamme de produits en intention plutôt qu'en extension. L'idée fondamentale est décrire les points communs entre les produits et d'offrir les constructions nécessaires (alternatives, options, choix de n parmi p, etc.) et des contraintes (exclusions mutuelles, dépendances, etc.) pour décrire correctement la variabilité parmi les produits. De cette façon, les concepteurs n'ont pas besoin d'énumérer toutes les configurations possibles. Cependant, la décomposition d'un système en points communs et points de variation nécessite des mécanismes efficaces et expressifs de composition pour pouvoir automatiquement dériver des configurations à partir de sélections de features.

Contribution (iia) au domaine de la gestion de la variabilité du logiciel [2,17,18,35,41,63,57,58,60, 67,68,69,88,90] Dans le cadre de la thèse de Bosco Joao Ferreira Filho nous avons contribué sur deux aspects directement liés à la modélisation de la variabilité. Tout d'abord, nous avons participé dans le cadre de la proposition de standard CVL pour Common Variability Language) à la définition de ce langage et à sa phase de standardisation. CVL vise à proposer un langage indépendant de tout domaine métier pour spécifier, capturer et résoudre la variabilité dans tout modèle instance d'un méta-modèle conforme au MOF[omg06]. CVL est constitué de trois parties importantes, le VAM (pour Variability Abstract Model), qui est la partie du CVL en charge d'exprimer la variabilité en termes d' une structure arborescente. Outre

le VAM, CVL contient également une partie réalisation de la variabilité capturée dans ce que l'on nomme le VRM (Variability Realization Model). Ce modèle permet de spécifier les changements à effectuer dans le modèle de base impliqué, selon les choix de résolution de la variabilité. Ces changements sont exprimés en points de variation du VRM. La dernière partie de CVL est composé du modèle de configuration permettant de définir les choix retenus pour une configuration particulière.

Dans le cadre de la thèse de Bosco, nous avons démontré le besoin de spécialisation de la sémantique de dérivation en fonction du domaine d'application dans lequel CVL est utilisé. Puis plus récemment, nous avons mis en évidence le besoin de définition de règles de cohérence entre le modèle métier, les informations du VAM et du VRM et nous avons proposé une méthode automatique pour la génération de contre exemples mettant en évidence le manque de règle métier permettant de contrôler la cohérence du triplet (VAm, VRM, Domaine métier) [1].

La communauté software composition (SC) (qui inclue notamment les communautés AOSD (*Aspect-Oriented Software Development*), FOSD (*Feature-Oriented Software Development*) et CBSD (*Component-Based Software Development*)) propose des mécanismes de compositions différents, mais complémentaires, tels que le tissage d'aspects, les mixins, la composition de features ou de composants, etc. Récemment, beaucoup d'approches proposent d'appliquer de tels mécanismes de composition dans le cadre des SPL "classiques" : niveau code [KAB07, AM04, ALS06, FCS+08, MO04] ou niveau modèle [LMV+07, JWEG07, GV08, MJBJ08, MBJ08, PKGJ08].

Contribution (iib) au domaine de la composition logicielle [3,4,19,23,27,32,33,34,41.45,49,71] Dans le cadre de la thèse de Mickael Clavreul, nous avons proposé une définition originale de la composition de modèles comme étant une paire correspondance-interprétation. Une correspondance définit la similarité entre deux modèles ou plus à partir d'un ensemble de règles d'alignement entre des « *patterns* »d'éléments de modèles. Une interprétation représente à la fois l'intention du processus de composition de modèles ainsi que les exigences de l'utilisateur en termes de sous-produits de la composition de modèles. Cette définition de la composition de modèles permet d'identifier des correspondances et des interprétations réutilisables pour spécifier une multitude de techniques de composition de modèles. A partir de cette définition, nous proposons un cadre théorique qui aide à (1) unifier les représentations des techniques existantes de composition de modèles et à (2) automatiser les processus de développement d'outils de composition de modèles dédiés. La pertinence et l'utilisabilité du cadre théorique impliquent la proposition de deux sous-contributions supplémentaires :

— Nous avons proposé un ensemble de catégories pour classer les techniques de correspondance entre modèles et les interprétations existantes de ces correspondances entre modèles. Ces catégories nous permettent de proposer une grille de lecture pour l'analyse et la comparaison des techniques existantes de composition de modèles.

— Nous avons proposé un langage de modélisation spécifique inspiré des catégories pour la définition de correspondances génériques entre modèles et pour la définition d'interprétations. Ce langage de modélisation outille la spécification et la construction de nouvelles approches pour la composition de modèles.

Les contributions proposées dans cette thèse ont été validées au travers de deux expérimentations principales : (i) les catégories de correspondances et d'interprétations ont été comparées en termes de précision et de pertinence à un ensemble significatif d'approches extraites de la littérature ; (ii)un prototype logiciel a été développé et utilisé dans le cadre du projet MOPCOM-I du pôle de compétitivité Images & Réseaux de la région Bretagne. La validation du langage de modélisation ainsi que l'approche globale de composition de modèles a été mise en œuvre sur un cas d'étude proposé par Technicolor pour l'intégration de librairies existantes dédiées à la gestion d'équipements numériques de diffusion vidéo.

Contribution (iic) : Évaluation de ces approches de composition et de synthèse de modèles sur les modèles utilisés à l'exécution [6,37,62]. Finalement ces nouveaux mécanismes de composition

1. ces derniers travaux ont reçu le prix du meilleur papier étudiant à la conférence SPLC (conférence majeur du domaine) et ce papier a été distingué comme un des trois meilleurs papiers de la conférence

logicielle ont été systématiquement validés sur l'utilisation des techniques de modélisation à l'exécution. En effet, Brice Morin dans le cadre de thèse a montré que la capacité à désynchroniser le modèle réflexif et la plate-forme est illustrée par la figure 1, permet d'exploiter à l'exécution ce modèle comme une cible pour tous les algorithmes qui synthétise à partir de diverses sources (*"feature model"* (dérivation de fonctionnalités), compositions d'aspects [MBNJ09]) ou composition de modèles, un modèle composé de l'architecture du das. Le système pouvant décider quand synchroniser le modèle, toutes les opérations avant déploiement *(≪off-fline≫)* ne sont pas contraintes par le système concret. Plusieurs travaux [MBJ+09],[KKR+12],[RBD+09] convergent vers l'utilisation de composant logiciel pour encapsuler les notions de cycles de vie des DAS et leurs opérateurs de composition. Utilisant divers paradigmes de conceptions et compositions pour assembler le système complet, la convergence de ces travaux étaye l'hypothèse à la fois d'une nécessité de diversité de paradigmes de composition, mais également de la viabilité d'exploiter ce niveau de granularité (composants) pour gérer les cycles de vie des briques applicatives des das. Au travers de cette thèse, nous avons démontré comment la complémentarité des approches de modélisation a permis la construction de configuration (architecture) correspondant à un ensemble de fonctionnalités, sans devoir spécifier toutes les configurations possibles à l'avance, tout en préservant un degré élevé de validation et le support de la reconfiguration automatiquement de DAS de sa configuration courante à une configuration nouvellement produite, sans devoir écrire à la main des scripts de reconfiguration de bas niveau et spécifiques à la plate-forme d'exécution ?

4.3.3 Contribution (iii) : Améliorations des techniques de modélisation pour leur utilisation à l'exécution [33]

Dans le cadre des travaux sur l'utlisation des modèles à l'exécution, une approche naturelle consiste à s'appuyer sur les defacto standard technique comme le cadre de modélisation proposé par Eclipse (*Eclipse Modeling Framework (EMF)*) [2]. Cependant, nous avons pu démontrer que de telles cadres de modélisation conçus pour le support des activités en phase de conception sont particulièrement inadaptés aux exigences d'une couche de modélisation persistante à l'exécution. Dans ce cadre, nous avons mis en évidence les exigences spécifiques pour la construction d'une couche de modélisation utilisable à l'exécution en pointant les lacunes des solutions existantes dans le domaine. Nous avons aussi présenté les contours d'une approche alternative. Cette alternative est très dynamique et propose le socle technique pour nos propositions d'améliorations des techniques de modélisation pour leur utilisation à l'exécution. Ces travaux ont donné lieu à deux publications [25] dont une est encore en révision au sein d'un journal. Un démonstrateur en ligne est aussi disponible [3].

4.4 Contribution (iv) : Models@runtime et distribution, [30,31,96]

Ces cinq dernières années, le nombre de types de terminaux qui permettent d'interagir avec ces systèmes d'information nouvelle génération a fortement augmenté notamment pour répondre aux besoins de mobilité : selon une étude de Gartner [Gar10], les ventes de Smartphones augmentent de 96% entre 2009 et 2010 pour atteindre 80 millions d'unités vendues par trimestre. Dans cette continuité, un ensemble de matériels regroupés sous l'appellation Internet des Objets est venu se greffer à ces systèmes complexes. Directement issu de l'électronique embarquée ces ≪capteurs/actionneurs≫ basse consommation connectent les systèmes informatiques à leur contexte physique. Les systèmes dédiés à l'hébergement des systèmes d'information eux-mêmes (DataCenters) connaissent également une phase de transition. L'émergence des techniques de mutualisation nommées sous le terme Cloud computing implique de repenser le développement de logiciel monolithique en système distribué. En effet afin de pouvoir profiter des services d'élasticité offerts par ces infrastructures, les logiciels doivent repenser les accès aux données, mais également la façon de séparer en sous-systèmes autonomes l'application en intégrant les contraintes imposées par le cycle de vie de ces sous-systèmes.

À ces nouveaux usages viennent s'ajouter les nouveaux modèles économiques. En effet si la loi de Moore (prédiction de l'évolution de la puissance de calcul) tend à ne plus s'appliquer, la durée de vie moyenne d'un matériel informatique tend à se réduire, rendant du même coup aussi court le temps d'obsolescence du logiciel tournant dessus (environ tous les ans pour les systèmes Android). Ce phénomène s'accentue d'autant pour tous les matériels basse consommation qui composent l'IOT. Ainsi comme il est expliqué par Ko *and al* [PCBD10], chaque cas d'application peut remettre en question la balance entre performance et

2. http ://www.eclipse.org/emf/
3. http ://kevoree.org/kmf/playground/

consommation et donc le choix du type de plate-forme.

La distribution, à savoir la répartition des constituants d'un système d'information sur les différents noeuds qui le composent est donc devenue inévitable et doit être prise en compte par les approches de génie logiciel. Là encore si ces besoins étaient il y a peu réservés à des élites de super calculateurs ou grilles de calcul il est aujourd'hui nécessaire pour des simples applications de gestion de prendre en compte ce besoin rendant là encore le besoin d'abstraction nécessaire pour la construction de ces systèmes.

La distribution des plates-formes induit de manière sous-jacente la nécessité de prendre en compte l'hétérogénéité de celles-ci.

Ce contexte d'exécution qui peut être rapproché d'une couche de réflexion impose donc d'avoir un état : une représentation globale ou partielle du système distribué qui permet alors de calculer le prochain état du das. La maintenance de la cohérence de cet état dans un environnement distribué est un problème difficile et particulièrement dans des environnements mobiles où les connexions sporadiques (intermittentes) rendent difficile la garantie de mise à jour de ces derniers. De manière opposée les noeuds d'un DataCenters doivent eux garantir des modifications coordonnées dans le même temps logique. Enfin dans un dernier exemple, dans le cas des réseaux de capteurs la synchronisation directe de la couche de réflexion a un impact très négatif sur la durée de vie de ces périphériques autonomes fonctionnant sur batterie. Ces trois cas de figure peuvent bénéficier d'une approche asynchrone de reconfiguration permettant à la fois de limiter les communications, mais également d'adopter différentes stratégies de dissémination du modèle d'architecture.

Contribution au travers de la thèse de François Fouquet En réponse à ce challenge, la thèse de François Fouquet confronte et étend le principe *Model@Runtime* pour définir, développer et déployer de façon continue les das distribués (ddas) sur des nœuds hétérogènes. Pour cela, cette thèse propose deux contributions principales :

- **Une couche générique de modélisation qui couvre de manière cohérente les grandes fonctionnalités d'un ddas**, à savoir : les nœuds de calculs, les composants métiers, les canaux de communication entre composants, ainsi que leurs capacités de synchronisation. Faisant le lien entre les outils de modélisation et les outils de développement, cette couche d'abstraction rend explicite l'hétérogénéité des implantations pour les différents types de fonctionnalité, depuis des composants déployés sur micro-contrôleurs[FBP+12] jusqu'à des infrastructures de serveurs.
- **L'introduction dans ce modèle d'une entité pour gérer la couche de réflexion distribuée et sa sémantique de dissémination**. En effet, en distribuant le modèle de réflexion du système distribué sur les différents nœuds qui le composent, il est alors possible de répondre au problème de partage de contexte nécessaire pour l'adaptation distribuée. Pour expliciter les différentes capacités de distribution et dissémination de ce modèle, l'approche propose une notion de groupe de nœuds qui encapsule cette sémantique. Inspirée des résultats d'algorithmique distribuée pour les réseaux pairs-à-pairs, la thèse de François Fouquet valide ce mode d'adaptation distribuée[FDP+12] en exploitant par exemple des approches dérivées du *gossip* [LTB+07],[EGKM04] pour l'adaptation sur réseau pairs-à-pairs. En associant différents groupes à des nœuds du système, on assure ainsi différentes propriétés de cohérence comme la consistance éventuelle [TTP+95], [BGL+06].

4.5 Validation sur différents domaines d'applications

Dans le cadre de la validation et de l'idée de l'utilisation du models@runtime pour la définition de systèmes dynamiques distribués et hétérogènes, nous avons évalué la pertinence de l'utilisation des approches fondées sur la notion de modèle à l'exécution dans plusieurs domaines métier à savoir les objets connectés et le cloud computing.

4.5.1 Contribution (v) : Models@runtime pour les objets connectés [39,45,46,47,36]

Face aux enjeux de société liés au vieillissement de la population, la domotique est souvent citée comme une solution pour favoriser le maintien à domicile des personnes âgées et la coordination des acteurs autour de cette problématique. Dans le cadre de cette thèse, nous avons caractérisé les exigences auxquelles doit faire face une plate-forme domotique. Nous avons identifié un ensemble de propriétés souhaitables pour un intergiciel orienté domotique, permettant le déploiement d'une solution à l'échelle d'une agglomération.

L'architecture d'un intergiciel a été définie et une expérience de déploiement de cet intergiciel dans le cadre du laboratoire domotique de l'université de Rennes 1 met en évidence la pertinence de la solution proposée. Cette thèse a donné lieu a un transfert sous la forme d'une tentative de création d'entreprise pour la commercialisation de ces résultats de recherche.

4.5.2 Contribution (vi) : Models@runtime pour le cloud computing [38,47,96]

Le Cloud Computing est devenu l'un des grands paradigmes de l'informatique et propose de fournir les ressources informatiques sous forme de services accessibles au travers de l'Internet. Ces services sont généralement organisés selon trois types ou niveaux. On parle de modèle SPI pour « *Software, Platform, Infrastructure* » en anglais.

De la même façon que pour les applications "standard", les services de Cloud doivent être capables de s'adapter de manière autonome afin de tenir compte de l'évolution de leur environnement. À ce sujet, il existe de nombreux travaux tels que ceux concernant la consolidation de serveur et l'économie d'énergie. Mais ces travaux sont généralement spécifiques à l'un des niveaux et ne tiennent pas compte des autres. Pourtant, comme l'a affirmé Kephart *et al.* en 2003 [KC03], même s'il existe des adaptations à priori indépendantes les unes des autres, celles-ci ont un impact sur l'ensemble du système informatique dans lequel elles sont appliquées. De ce fait, une adaptation au niveau infrastructure peut avoir un impact au niveau plate-forme ou au niveau application.

Dans le cadre de la thèse d'Erwan Daubert encadrée en collaboration avec Françoise André puis Jean-Louis Pazat, nous avons cherché à évaluer la pertinence de l'utilisation de l'idée du models@runtime afin de fournir un support pour l'adaptation permettant de gérer celle-ci comme une problématique transverse aux différents niveaux du cloud computing et dans le but d'assurer la cohérence et l'efficacité de l'adaptation. Pour cela, cette thèse a proposé une extension à l'abstraction utilisée habituellement dans la cadre du models@runtime capable de représenter l'ensemble des niveaux et servant de support pour la définition des reconfigurations. Afin de montrer l'utilisabilité de cette extension, nous avons présenté trois expérimentations permettant de montrer l'extensibilité et la généricité de notre solution, de montrer le faible impact sur les performances du système, lié à l'utilisation du models@runtime, et de montrer que cette abstraction permet de faire de l'adaptation multi-niveaux.

4.5.3 Contribution (viii) : Kevoree [96]

Ces propositions sont synthétisées dans une plateforme logicielle nommée Kevoree (www.kevoree.org), qui combine une approche Model@Runtime et un modèle de composant pour la modélisation complète des éléments d'un système dynamique distribué et hétérogène. cette plateforme logicielle open-source utilisée en enseignement et dans le cadre de projets collaboratifs par les partenaires est une des trois plateformes majeures de la proposition de la nouvelle équipe de recherche Diverse qui vise à prendre la suite de Triskell.

4.6 Perspectives de Recherche

De nombreuses perspectives de recherches sont encore ouvertes dans la suite de ces travaux. Dans ce cadre, plusieurs projets européens et plusieurs projets nationaux vont permettre de consolider les propositions, transférer les idées, mais aussi évaluer de nouvelles idées parmi lesquelles :

4.6.1 Models@runtime pour le web

Dans le cadre d'un projet avec la société Zenexity, acteur majeur dans l'ingénierie des applications Web grâce à son cadre de développement logiciel play [4], et au travers de la thèse de Julien Richard Foy nous menons différentes études pour comprendre les bonnes abstractions utilisables dans le domaine du web aussi bien d'un point de vue architecture que d'un point de vue langage de développement. Dans ce cadre, un premier résultat publié à la conférence GPCE 2013 [14] a montré comment la définition d'abstractions propres au web pouvait être compilée efficacement aussi bien en cas d'exécution coté serveur ou côté client.

4. http ://www.playframework.com/

Dans les perspectives, nous travaillons afin de trouver les bons mécanismes pour un support efficace des développements d'applications web résilientes.

4.6.2 Models@runtime pour le contrôle de la consommation de ressources dans un environnement virtualisé

Dans le cadre du projet ANR InfraJVM, et de la thèse d'Inti Gonzalez Herrera dont je suis directeur, et que je co-encadre avec Johann Bourcier, nous proposons d'étudier la définition d'abstraction pour la modélisation des données liées à la consommation des ressources d'une application que ce soit le réseau, la mémoire ou la consommation d'énergie. Pour ce faire, nous travaillons à la fois sur la définition d'une telle abstraction, mais aussi sur les mécanismes permettant un monitoring peu gourmand de la consommation des ressources. Enfin, nous travaillons à la définition d'heuristiques permettant de se servir des méta-données disponibles dans la couche de modélisation à l'exécution afin de détecter de manière opportuniste des déviations sur la consommation des ressources par un composant.

4.6.3 Models@runtime pour le test de systèmes distribués

Dans le cadre du projet européen Heads, et de la thèse de Mohamed Boussa qui a démarré en octobre 2013, nous proposons d'étudier comment l'utilisation du *models@runtime* peut permettre le test de systèmes distribués et hétérogènes à coût raisonnable. La problématique est d'offrir des capacités de vérification d'un certain nombre de propriétés d'un système dynamique distribué et hétérogène à coup faible pour le concepteur de tels systèmes en synthétisant à la demande et de manière maligne l'infrastructure, les données et les jeux de tests pour un tel système.

4.6.4 Synthèse de modèles et diversité

Enfin dans le cadre du projet européen FET Diversify dirigé par Benoît Baudry et dans le cadre de la thèse de yeboah-antwi-kwaku qui a débuté en octobre 2013, nous proposons d'étudier la notion de diversité qui peut être utilisée comme fondement d'un principe nouveau de conception de logiciels et l'augmentation des capacités d'adaptation des systèmes dits ouverts. L'augmentation de la diversité dans le système vise à permettre au logiciel de s'adapter à des situations imprévues au moment de la conception. L'approche scientifique du projet diversify et de cette thèse est basée sur une forte analogie avec les systèmes écologiques, la biodiversité et l'évolution. DIVERSIFY réunit des chercheurs des domaines des systèmes distribués à logiciel prépondérant et de l'écologie afin de traduire les concepts et les processus écologiques dans les principes de conception de logiciels.

4.6.5 Transfert

L'ensemble de ces perspectives de recherche est actuellement supporté soit par un contrat direct avec une entreprise soit par la participation à un projet européen (Diversify ou Heads). Le souci d'ancrer mes perspectives de recherche dans le cadre de projets collaboratifs vise à garantir les ressources sur ces perspectives mais aussi cela permet de confronter nos idées sur des cas réels issus d'industriels partenaires.

Références

[ALS06] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers : aspects and features in concert. In *ICSE '06 : Proceedings of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM.

[AM04] M. Anastasopoulos and D. Muthig. An evaluation of aspect-oriented programming as a product line implementation technology. In *ICSR'04 : 8th International Conference on Software Reuse : Methods, Techniques and Tools*, pages 141–156, Madrid, Spain, 2004.

[BBF09] G. Blair, N. Bencomo, and R.B. France. Models@ run.time. *Computer*, 42(10) :22 –27, oct. 2009.

[BBFS08] N. Bencomo, G. Blair, C. Flores, and P. Sawyer. Reflective Component-based Technologies to Support Dynamic Variability. In *VaMoS'08 : 2nd Int. Workshop on Variability Modeling of Software-intensive Systems*, Essen, Germany, January 2008.

[BGFB08] N. Bencomo, P. Grace, C. Flores, and D. Hughesand G. Blair. Genie : Supporting the Model Driven Development of Reflective, Component-based Adaptive Systems. In *ICSE 2008 - Formal Research Demonstrations Track*, 2008.

[BGH+06] J. Bayer, S. Grard, O. Haugen, J. Mansell, B. Moller-Pedersen, J. Oldevik, P. Tessier, J.-P. Thibault, and T. Widen. *Software Product Lines*, chapter Consolidated Product Line Variability Modeling, pages 195–242. Number ISBN : 978-3-540-33252-7. Springer Verlag, 2006.

[BGL+06] R. Baldoni, R. Guerraoui, R. Levy, V. Quema, and S. Piergiovanni. Unconscious eventual consistency with gossips. *Stabilization, Safety, and Security of Distributed Systems*, pages 65–81, 2006.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. CLOS in context : the shape of the design space. pages 29–61, 1993.

[BS06] J. Boardman and B. Sauser. System of systems - the meaning of of. *System of Systems Engineering*, 0 :6 pp.–, 2006.

[CLG+09] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems : A research roadmap. 5525 :1–26, 2009.

[CN01] P. Clements and L. Northrop. *Software product lines : practices and patterns*, volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[EGKM04] P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE computer*, 37(5) :60–67, 2004.

[FBP+12] François Fouquet, Olivier Barais, Noël Plouzeau, Jean-Marc Jézéquel, Brice Morin, and Franck Fleurey. A Dynamic Component Model for Cyber Physical Systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, Bertinoro, Italie, July 2012.

[FCS+08] Eduardo Figueiredo, Nélio Cacho, Claudio SantAnna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco Dantas. Evolving software product lines with aspects : an empirical study on design stability. In *ICSE'08 : 30th International Conference on Software Engineering*, pages 261–270, Leipzig, Germany, may 2008. ACM.

[FDP+12] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *DAIS 2012*, Stockholm, Suède, June 2012.

[FS09] F. Fleurey and A. Solberg. A Domain Specific Modeling Language supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *MODELS'09 : ACM/IEEE 12th International Conference on Model-Driven Engineering Languages and Systems*, Denver, Colorado, USA, oct 2009.

[Gar10] Gartner. Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010 ; smartphone sales increased 96 percent, 2010. http ://www.gartner.com/it/page.jsp ?id=1466313.

[GBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *WICSA '01 : Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, page 45, Washington, DC, USA, 2001. IEEE Computer Society.

[GF99] R. Guerraoui and M.E. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4) :101–104, 1999.

[Got08] Greg Goth. Ultralarge systems : Redefining software engineering ? *IEEE Software*, 25 :91–94, 2008.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[GV08] I. Groher and M. Voelter. Using Aspects to Model Product Line Variability. In *EA@SPLC'08 : 13th International Workshop on Early Aspects at SPLC*, Limerick, Ireland, 2008.

[HGC+06] Danny Hughes, Phil Greenwood, Geoff Coulson, Gordon Blair, Florian Pappenberger, Paul Smith, and Keith Beven. Gridstix : : Supporting flood prediction using embedded hardware and next generation grid middleware. In *4th International Workshop on Mobile Distributed Computing (MDC'06)*, Niagara Falls, USA, 2006.

[HHPS08] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4), April 2008.

[HSSF06] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using product line techniques to build adaptive systems. In *SPLC'06 : 10th Int. Software Product Line Conf.*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[JWEG07] P.K. Jayaraman, J. Whittle, A.M. Elkhodary, and H. Gomaa. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS'07 : 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, Oct 2007.

[KAB07] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *SPLC '07 : 11th International Software Product Line Conference*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003.

[KKR+12] JeongGil Ko, Kevin Klues, Christian Richter, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low power or high performance ? a tradeoff whose time has come (and nearly gone). In Gian Picco and Wendi Heinzelman, editors, *Wireless Sensor Networks*, volume 7158 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2012.

[LMV+07] Ph. Lahire, B. Morin, G. Vanwormhoudt, A. Gaignard, O. Barais, and J. M. Jézéquel. Introducing Variability into Aspect-Oriented Modeling Approaches. In *MoDELS'07 : 10th Int. Conf. on Model Driven Engineering Languages and Systems*, Nashville USA, October 2007.

[LTB+07] S. Lin, F. Taiani, G.S. Blair, et al. Gossipkit : A framework of gossip protocol family. In *Proceedings of the 5th MiNEMA Workshop (Middleware for Network Eccentric and Mobile Applications)*, pages 26–30, 2007.

[MBJ08] B. Morin, O. Barais, and J. M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *VaMoS'08 : 2nd Int. Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, January 2008.

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10) :44–51, 2009.

[MBNJ09] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.

[McI68] M.D. McIlroy. Mass produced software components. In P. Naur and Randell B., editors, *Proceedings of the NATO Conference on Software Engineering*, pages 138–155, Garmish, Germany, October 1968. NATO Science Committee.

[MJBJ08] B. Morin, J.Klein, O. Barais, and J. M. Jézéquel. A Generic Weaver for Supporting Product Lines. In *EA@ICSE'08 : Int. Workshop on Early Aspects*, Leipzig, Germany, May 2008.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT'04/FSE-12 : 12th ACM SIGSOFT international symposium on Foundations of Software Engineering*, pages 127–136, Newport Beach, CA, USA, 2004. ACM.

[Mor10] Brice Morin. *Leveraging Models from Design-time to Runtime to Support Dynamic Variability.* PhD thesis, Université Rennes 1, Septembre 2010.

[MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECCOP '98 : Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 355–382. Springer-Verlag, 1998. *ISBN : 3-540-64737-6.*

[NDBJ08] Grégory Nain, Erwan Daubert, Olivier Barais, and Jean-Marc Jézéquel. Using mde to build a schizophrenic middleware for home/building automation. In *Proceedings of the 1st European Conference on Towards a Service-Based Internet*, ServiceWave '08, pages 49–61, Berlin, Heidelberg, 2008. Springer-Verlag.

[OGT+99a] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications, IEEE*, 14(3) :54–62, 1999.

[OGT+99b] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3) :54–62, 1999.

[omg06] omg. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[PCBD10] Carlos Parra, Anthony Cleve, Xavier Blanc, and Laurence Duchien. Feature-based composition of software architectures. In Muhammad Babar and Ian Gorton, editors, *Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 230–245. Springer Berlin / Heidelberg, 2010.

[PKGJ08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi, and Jean-Marc Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *SPLC'08 : 12th International Software Product Line Conference*, pages 339–348, Limerick, Ireland, September 2008. IEEE Computer Society.

[RBD+09] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. Music : Middleware support for self-adaptation in ubiquitous and service-oriented environments. In Betty Cheng, RogÃ©rio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 164–182. Springer Berlin / Heidelberg, 2009.

[RNC+95] S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial intelligence : a modern approach*. Prentice hall Englewood Cliffs, NJ, 1995.

[RWLN89] Jeff Rothenberg, Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen. The Nature of Modeling. In *in Artificial Intelligence, Simulation and Modeling*, pages 75–92. John Wiley & Sons, 1989.

[Sto09] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE '09.*, pages 369 –374, aug. 2009.

[Szy96] C. Szyperski. Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, 1996.

[TTP+95] D.B. Terry, M.M. Theimer, K. Petersen, A.J. Demers, M.J. Spreitzer, and C.H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review*, 29(5) :172–182, 1995.

[Ver10] VersionOne. State of agile development, 2010. http ://www.versionone.com/pdf/2011_State_of_Agile_Dev

[ZC06] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM.

[ZJ06] Tewfik Ziadi and Jean-Marc Jézéquel. *Software Product Lines*, chapter Product Line Engineering with the UML : Deriving Products, pages 557–586. Number ISBN : 978-3-540-33252-7. Springer Verlag, 2006.

5 Activités d'encadrement

Co-encadrement de thèse

De 2007 à 2010, j'ai co-encadré la thèse de **Brice Morin** à hauteur de 50% avec Jean-Marc Jézéquel. La contribution de cette thèse a été de permettre une meilleure intégration des modèles de conception à l'exécution, ceci afin de bénéficier de tous les travaux sur la transformation de modèles afin d'adapter les applications à leur contexte d'exécution.

De septembre 2008 à décembre 2011, j'ai co-encadré la thèse de **Grégory Nain** à hauteur de 50% avec Jean-Marc Jézéquel. La contribution de cette thèse a été de proposer une méthodologie pour la construction d'applications à base de services dans le domaine de la domotique.

De Octobre 2008 à décembre 2011, j'ai co-encadré la thèse de **Mickaël Clavreul** à hauteur de 50% avec Jean-Marc Jézéquel. La contribution de cette thèse a été de travailler sur une algèbre pour la composition de modèles.

Depuis Octobre 2009 à mai 2013, j'ai co-encadré la thèse de **Erwan Daubert** à hauteur de 50% avec Françoise André malheureusement décédé en 2011. Jean-Louis Pazat a pris la suite de la direction de la thèse. L'objectif de cette thèse est de travailler sur la planification et l'exécution d'adaptations transverses aux couches d'application de cloud.

De Octobre 2009 à Mars 2013, j'ai participé activement à l'encadrement de la thèse de **François Fouquet** avec Jean-Marc Jézéquel et Noël Plouzeau. L'objectif de cette thèse est de travailler sur la problématiques d'adaptation logicielle à l'aide de modèles à l'exécution pour des systèmes hétérogènes et hautement distribués.

De Octobre 2009 à Février 2013, j'ai participé à l'encadrement de la thèse de **Paul Istoan** avec Jean-Marc Jézéquel thèse en co-tutelle avec l'université du Luxembourg. L'objectif de cette thèse est de travailler sur la problématique de la variabilité logicielle.

Depuis Octobre 2010, je participe activement à l'encadrement de la thèse d'**Emmanuelle Rouillé** avec Jean-Marc Jézéquel et Benoit Combemale. L'objectif de cette thèse est de travailler sur la problématique de capitalisation dans les processus d'entreprise.

Depuis Octobre 2011, je co-encadre la thèse de **Joao Bosco Ferreira Filho** à hauteur de 70% avec Benoit Baudry. L'objectif de cette thèse est de travailler sur une sémantique optionnelle d'un opérateur de dérivation pour les lignes de produit en ingénierie système multi-vues.

Depuis Octobre 2011, je co-encadre la thèse de **Julien Richard Foy** à hauteur de 70% avec Jean-Marc Jézéquel. L'objectif de cette thèse est de travailler sur un canevas de langages dédiés pour la création efficace et sure d'applications web.

Depuis Septembre 2012, je co-encadre la thèse de **Inti Gonzales Herreira** en tant que directeur sur dérogation pour absence d'HDR à hauteur de 50% avec Johann Bourcier. L'objectif de cette thèse est de travailler sur la construction de machines virtuelles adaptées aux applications construites à base de composants et s'exécutant dans un environnement contraints.

Depuis Octobre 2013, je co-encadre la thèse de **Thomas Degueule** en tant que directeur sur dérogation pour absence d'HDR à hauteur de 50% avec Arnaud Blouin. L'objectif de cette thèse est de travailler sur la modularité dans la construction de langages.

Depuis Novembre 2013, je co-encadre la thèse de **Mohammed Boussa** à hauteur de 80% avec Benoit

Baudry. L'objectif de cette thèse est de travailler sur le test automatisé de générateurs de code dans un contexte d'ingénierie dirigée par les modèles.

Encadrement de stages de Master Recherche

De Octobre 2008 à Juin 2009, j'ai co-encadré avec Régis Fleurquin le stage effectué à l'INRIA de Mohammed Said Belaid originaire du Master Recherche de l'Université de Rennes 1. Mohammed a travaillé sur l'étude de l'évaluation incrémentale de contrainte OCLs ndans un contexte de Model Driven Engineering.

De Octobre 2009 à Juin 2010, j'ai co-encadré avec Jean-Marc Jézéquel le stage effectué à l'INRIA de Antonio Neto originaire du Master Recherche de l'Université de Rennes 1. Antonio a travaillé sur l'inférence de type de modèles à partir d'un jeu de contraintes OCL dans un contexte de Model Driven Engineering.

Autres stagiaires

J'ai assuré le suivi de stages de fin d'études dans les différentes formations de Master de l'ISTIC (10 stagiaires par an). Le suivi consiste à veiller au bon déroulement du stage, à visiter le stagiaire dans l'entreprise, à évaluer son rapport de stage et à organiser sa soutenance.

6 Rayonnement scientifique

Jury de thèse

En dehors des doctorants que j'ai eu la chance de co-encadrer, j'ai été membre du jury pour les thèses suivantes en tant qu'examinateur.

- Pankesh Patel - Environnement de développement d'applications pour l'internet des objets - Université Pierre et Marie Curie soutenue le 26 novembre 2013.
- Ali Hassan - Adding Spatial Information to Software Component Model - The Localization Effect - Université européenne de Bretagne soutenue le 24 septembre 2012
- Mohamad Fady Hamoui - Un système multi-agents à base de composants pour l'adaptation autonomique au contexte - Application à la domotique - Université de Montpellier 2, soutenue le 13 décembre 2010.
- Jérémy DUBUS - Une démarche orientée modèle pour le déploiement de systèmes en environnements ouverts distribués - Université de Lille 1, soutenue le 10 octobre 2008.

Collaborations pédagogiques

Outre, l'enseignement dans différentes écoles du grand ouest, j'ai eu aussi l'occasion :

- de fournir une expertise sur l'évaluation du contenu pédagogique du département ISIC des Mines de Douai,
- de participer en tant que formateur à des stages de perfectionnement à destination de professeurs de BTS IRIS et SIO en 2010,
- d'être orateur au sein d'une école d'été dans le domaine du *service oriented computing* en Crète en 2010,
- d'être orateur deux années consécutives pour l'école des jeunes chercheurs en programmation. Formation d'une dizaine de jours à destination des doctorants du GDR-GPL.

Comités de programmes

J'ai été membre du comité de programme de la conférence IDM'2009, de la conférence LMO'09, des conférences CIT'09 et CIT'10, des conférences Euromicro 2011, 2012, 2013 2014 des conférences CIEL'12, CIEL'13, CIEL'14 des conférences CARI'09, CARI'10 et CARI'11, de la conférence HPCC'2014, de l'atelier Splat'14, des conférences CAL'09, CAL'10, CAM'14 de la conférence WETICE 13, de la conférence SAC 2013 et de la conférence SC'2013 des ateliers CAMPUS'09, CAMPUS'10 et CAMPUS'11, des ateliers VariComp 2010, 2011 et 2012, 2013 de l'atelier MetaAspect'10, des ateliers VAO 2013 et 2014, SAC'2014, des démos d'outils de la conférence CSMR'13.

Activité de relecture

En 2009, 2010, 2011 ET 2012 j'ai été relecteur pour les revues *Software and Systems Modeling*, *Transactions on Software Engineering*, *Transactions on Aspect-Oriented Software Development*, *Journal of Systems and Software*, *Transaction on the Web*, *IEEE Software*, *IEEE Computer* and *Software Practise and Experience*. J'ai aussi été relecteur pour les conférences ICSE'09, ICSE'10, ICSE'11 et ICSE'12, AOSD'09, ECOOP'11, Models'09, Models'10, Model'11, Model'12, SASO'12, Vamos'11, Vamos'12, SC'12, Models'13, Models'14, SPLC'14, SLE'14, CBSE'14.

Développement logiciel

Je participe activement au développement de la plate-forme kevoree [5]. Kevoree fournit une plate-forme logicielle fondée sur le principe de l'utilisation de modèles à l'exécution (models@runtime) pour la construction d'applications adaptables s'exécutant dans un environnement hétérogène et distribué. La prolifération d'équipements autour de la mouvance de l'Internet des objets amène une grande diversité des usages et des plates-formes d'exécutions. Ces usages exploitent de manière croissante le caractère distribué et autonomique de ces équipements amenant une vraie complexité de développement. La réflexion nécessaire à ces usages afin de profiter de ces équipements de manière optimale devient alors une préoccupation majeure. L'approche Models@Runtime vise justement à encadrer et étendre la gestion de cette couche de réflexion distribuée. Kevoree est une approche outillée autour de ce paradigme pour répondre à des enjeux d'ingénierie logicielle telle que la construction d'un système tactique de terrain pour les opérations d'urgence.

Je participe aussi activement au développement d'outils pour le langage de méta-modélisation Kermeta [6]. La méta-modélisation dans le domaine de l'informatique se définit comme la mise en évidence d'un ensemble de concepts pour un domaine particulier. Un modèle est une abstraction d'un phénomène du monde réel tandis qu'un méta-modèle est une abstraction d'ordre supérieur mettant en évidence les concepts utilisés pour définir le modèle. Dans ce domaine, on peut parler d'un modèle conforme à son méta-modèle comme on peut parler d'un programme conforme à sa grammaire. L'OMG au travers du MOF [7] propose un langage standardisé afin de permettre la définition de nouveaux méta-modèles. Kermeta, développé au sein de l'équipe Triskell, est un langage de méta-modélisation exécutable construit comme une extension du MOF. Il permet de définir un méta-modèle auquel on associe une sémantique opérationnelle pour permettre la simulation des modèles. Le MOF ainsi que Kermeta permettent de définir un certain nombre de contraintes sur le modèle représenté, portant entre autres sur la cardinalité des relations entre les concepts. Ces contraintes peuvent servir à vérifier la cohérence d'un modèle par rapport à son méta-modèle. J'ai particulièrement travaillé à l'intégration du langage OCL à la plateforme Kermeta ainsi qu'à la mise en œuvre du compilateur.

5. http ://www.kevoree.org
6. http ://www.kermeta.org
7. http ://www.omg.org

7 Animation scientifique

Participation à l'organisation de conférences et de manifestation

— J'ai été **président** du comité d'organisation de la conférence CIEL et des troisièmes journées du GDR-GPL. Ces manifestations ont réuni 220 personnes à Rennes en juin 2012. J'ai coordonné pour cette manifestation avec Elisabeth Lebret à la fois la logistique et les relations avec le président des comité de programmes en charge des contenus scientifiques.

— Depuis 2012, je participe modestement avec un collègue Didier Certain à l'organisation à l'organisation des conférences du BreizJug au sein de l'université. Nous fournissons principalement un relai logistique au sein de l'université. Cette association regroupant des passionnées de la communauté Java Rennaise propose une conférence technique chaque premier lundi du mois. En outre, je participe aussi modestement à l'organisation du BreizhCamp en fournissant un support logistique côté université. Le BreizhCamp est une manifetation de 2 jours regroupant environ 200 personnes autour de conférences techniques sur les problématiques du développement d'applications.

— J'ai organisé avec Benoit Combemale une journée sur la modélisation pour fédérer les acteurs bretons le 9 septembre 2010 à l'IRISA. 32 chercheurs académiques ont participé à cette journée.

— Je suis intervenu en 2009 et 2011 dans le cadre de l'école des jeunes chercheurs en programmation (EJCP).

— J'ai organisé la deuxième journée autour du langage Kermeta qui s'est déroulée le 10 Octobre 2006 à l'IRISA. 72 chercheurs académiques ou industriels ont participé à cette journée.

— J'ai organisé avec Benoit Baudry une journée sur la modélisation par aspects le 8 juin 2007 à l'IRISA. 32 étudiants ou chercheurs académiques ont participé à cette journée.

— J'ai organisé la troisième journée autour du langage Kermeta qui s'est déroulée le 6 décembre 2007 à l'IRISA. 84 étudiants, chercheurs académiques ou industriels ont participé à cette journée.

— J'ai organisé sur le site de Rennes la nuit de l'info 2007 http://www.nuitdelinfo.com/. Des groupes d'étudiants de différentes formations (sur différents sites) ont relevé les défis que leur ont lancé des entreprises et des organisations pour réaliser une application informatique complexe dans un temps limité (entre le coucher et le lever du Soleil).

— Pendant ma thèse à Lille, j'ai participé à l'organisation de différentes conférences organisées par l'équipe dont JFPLA'05.

8 Ouverture sur le monde industriel

— De 2007 à 2008, j'ai monté et suivi le projet SintArch, projet de collaboration entre l'université de Pernambouc et INRIA.

— De 2008 à 2010, j'ai travaillé sur l'Action Développement Technologique INRIA Galaxy [8].

— De 2008 à 2011, j'ai travaillé dans le cadre de la thèse de Brice Morin sur le projet STREP FP7 nommé DiVA (Dynamic Variability In Complex, Adaptive Systems).

— De 2008 à 2011, j'ai travaillé sur le projet ANR Harpege MOVIDA.

— De 2008 à 2011, j'ai travaillé sur le projet du pôle de compétitivité Image et Réseaux MOPCOM-I [9].

— De 2008 à 2012 j'ai participé au réseau d'excellence européen s-cube [10].

— De 2009 à 2011, j'ai travaillé sur l'Action Développement Technologique INRIA DAUM.

— Depuis octobre 2011, je coordonne un projet de collaboration direct avec Thales TRT nommé VaryMDE. Ce projet d'une durée de 42 mois vise à travailler sur les problématiques de conception système dans un contexte d'ingénierie multi-vue.

— De 2012 à 2013, je participe à un projet avec le musée de Bretagne qui vise à créer un projet de jeu sur la métropole Rennaise afin d'inciter les jeunes à venir au musée de Bretagne.

— A partir de décembre 2012, je travaille sur le projets européen Merge en tant que coordinateur pour l'INRIA.

— Depuis 2012, je participe activement au projet ANR Gemoc que Benoit Combemale, membre de l'équipe Triskell, coordonne.

— Depuis 2013, je participe activement au projet FET Diversify que Benoit Baudry, responsable de l'équipe Triskell, coordonne.

— Depuis septembre 2013, je travaille sur le projet européen HEADS en tant que coordinateur pour l'INRIA et WorkPackage leader.

— Depuis novembre 2013, je travaille à 20% dans le cadre de l'institut de Recherche Technologique B-COM. Je suis associé au sein du projet INDEED.

— A partir de septembre 2014 je travaillerai sur le projet FUI Clarity.

Le projet européen Strep Heads (*Heterogeneous and Distributed Service for the Future Computing Continuum*) vise à établir de nouvelles méthodologies de conception logiciel pour les applications internet futures qui offre les caractéristiques suivantes (système massivement distribué, hétérogène, ouvert dont les évolutions de l'infrastructure ou des exigences sont non prévisibles). (10/2013- 10/2016).

Le cloud permet au plus grand nombre (entreprises, individus) de disposer de capacité de calcul, de partage des données, d'accessibilité, ... Pour accompagner cette révolution en marche, le projet INDEED de l'IRT B-COM veille à optimiser la localisation des données et garantir leur protection en tenant compte de l'impact écologique des data centers. (10/2013- 10/2016).

Le projet européen Strep Heads (*Heterogeneous and Distributed Service for the Future Computing Continuum*) vise à établir de nouvelles méthodologies de conception logiciel pour les applications internet futures qui offre les caractéristiques suivantes (système massivement distribué, hétérogène, ouvert dont les évolutions de l'infrastructure ou des exigences sont non prévisibles). (10/2013- 10/2016).

8. http://galaxy.gforge.inria.fr/
9. http://www.mopcom-i.org
10. http://www.s-cube-network.eu/

Le projet européen FET Diversify explore comment la diversité peut être utilisée comme fondement d'un principe nouveau de conception de logiciels et l'augmentation des capacités d'adaptation des systèmes dits ouverts. L'augmentation de la diversité dans le système vise à permettre au logiciel de s'adapter à des situations imprévues au moment de la conception. L'approche scientifique du projet Diversify est basée sur une forte analogie avec les systèmes écologiques, la biodiversité et l'évolution. Diversify réunit des chercheurs des domaines des systèmes distribués à logiciel prépondérant et de l'écologie afin de traduire les concepts et les processus écologiques dans les principes de conception de logiciels. (Durée 36mois : 03/13-03/16)

Le projet ITEA Merge démarré en décembre 2012 vise à la construction d'un environnement d'ingénierie système de modélisation mutli-vues. Dans la suite du projet Movida, le projet Merge nous permet d'intégrer nos travaux sur la variabilité dans un environnement de modélisation multi-vues et nos travaux sur les architectures à composants. (Durée 36 mois : 03/12-12/15)

Le projet INS Gemoc, démarré en décembre 2012 vise la définition d'un *framework* de modèles de calcul génériques pour l'exécution et l'analyse dynamique de modèles. (12/12-05/16)

Le projet de collaboration direct VaryMDE, projet monté dans la continuité du projet collaboratif Movida, vise la poursuite de nos travaux sur la variabilité dans un contexte mutli-vue d'ingénierie de modélisation système. Ce projet a pour but de faciliter le transfert de nos activités de recherche vers Thales (Durée 42mois : 09/11-3/15)

Le projet ANR Harpege MOVIDA démarré en Janvier 2008 vise à travailler sur un atelier d'architecture pour le support à la modélisation de vues et l'aide à la décision pour les architectes dans un contexte de conception de systèmes de systèmes. (Durée 3ans : 01/09-12/11)

Le projet DGE Mopcom-I : Ingénierie dirigée par les modèles pour la spécification et la conception de système par la fiabilisation des processus basés sur les modèles vise à travailler sur un processus fiable basé sur les modèles dans un contexte de conception de systèmes de systèmes. (Durée 3ans : 10/08-09/11)

le réseau d'excellence européen s-cube réunit les différentes équipes travaillant dans le domaine de la conception de logiciels et les approches orientées services. Il a pour objectif de permettre une meilleure intégration des travaux existants et proposer un pôle d'expertise européen dans le domaine des services. (Durée 4ans : 03/08-02/12)

De 2008 à 2011, j'ai travaillé dans le cadre de la thèse de Brice Morin sur le projet STREP FP7 nommé DiVA (Dynamic Variability In Complex, Adaptive Systems). Ce projet visait à fournir une nouvelle méthodologie outillée avec un *framework* intégré pour la gestion dynamique de la variabilité dans les systèmes adaptatifs. Diva a proposé de combiner les approches par aspects et les techniques basées sur les modèles pour atteindre cet objectif. (Durée 3ans : 03/08-02/11)

De 2008 à 2010, j'ai travaillé sur les Action Développement Technologique Galaxy [11]. L'ADT (Action de Développement Technologique INRIA) galaxy vise le développement d'un environnement intégré permettant à INRIA, autour des architectures logicielles agiles et dynamiques, de se positionner dans le domaine des « architectures orientées services » (ou SOA, Service-Oriented Architecture). (Durée 2ans : 10/08-09/10)

J'ai en outre participé au montage du projet SintArch (projet FACEPE) avec l'université de Pernambouc au Brésil. Le projet SintArch a pour but de travailler sur l'intégration des informations de coordination dans une architecture à base de composants. Ce projet qui a démarré en Mars 2007 sera l'occasion d'augmenter le partenariat entre l'IRISA et l'université du Pernambouc. Ce projet s'est terminé en Février 2009.

11. http://galaxy.gforge.inria.fr/

9 Publications et production scientifique

Revue

[1] Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher, Jérôme Le Noir, Axel Legay and Benoit Baudry. **Generating Counterexamples of Model-based Software Product Lines**, dans *Journal on Software Tools for Technology Transfer (STTT)*, Springer 341, (2014)

[2] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus, François Fouquet, **Mashup of metalanguages and its implementation in the Kermeta language workbench**, dans Software and Systems Modeling (SoSyM), dans *Software and Systems Modeling (SoSyM)*,(2013)

[3] Gilles Perrouin, Gilles Vanwormhoudt, Brice Morin, Philippe Lahire, Olivier Barais, Jean-Marc Jézéquel, **Weaving Variability into Domain Metamodels**, dans Software and Systems Modeling (SoSyM), dans *Software and Systems Modeling (SoSyM)*,dec.(2012)

[4] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, et Jean-Marc Jézéquel (2012) **Reusable model transformations**. dans *Software and Systems Modeling (SoSyM)*, tba :346-379.(2012)

[5] Naouel Moha , Sagar Sen, Cyril Faucher, Olivier Barais et Jean-Marc Jézéquel **Evaluation of Kermeta for solving graph-based problems**. dans *International Journal on Software Tools for Technology Transfer (STTT)*, 12(3–4) :273-285. (2010)

[6] Franck Chauvel , Olivier Barais , Isabelle Borne et Jean-Marc Jézéquel **Un processus à base de modèles pour les systèmes auto-adaptatifs**. Revue de l'Electricité et de l'Electronique (REE) (2009).

[7] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey et Arnor Solberg, Models at Runtime to Support Dynamic Adaptation. IEEE Computer. p46–53, (2009)

[8] Sébastien Saudrais, Olivier Barais, Laurence Duchien et Noel Plouzeau. **From formal specifications to QoS monitors**. dans *Journal of Object Technology, Special Issue on Advances in Quality of Service Management*, 2007, 6(11) :7–24.

[9] Olivier Barais, Philippe Lahire, Alexis Muller, Noël Plouzeau et Gilles Vanwormhoudt. **Évaluation de l'apport des aspects, des sujets et des vues pour la composition et la réutilisation des modèles**. dans *la Revue RSTI-L'Objet. Hermès Sciences.* 2007 2-3/2007(13) :177–212.

[10] Olivier Barais, Alexis Muller et Nicolas Pessemier. **Vers une Séparation Entités/Fonctions au sein d'une Architecture Logicielle à base de Composants.** dans *la Revue RSTI-L'Objet. Hermès Sciences.* 2005 2-3/2007(13) :115–139.

[11] Olivier Barais, Laurence Duchien. **TranSAT : maîtriser l'évolution d'une architecture logicielle.** dans *la Revue RSTI-L'Objet. Hermès Sciences.* 2004 2-3/2004(10) :103–116.

Book Chapter

[12] Jean-Marc Jézéquel, Olivier Barais, et Franck Fleurey. **Model Driven Language Engineering with Kermeta**. Dans *3rd Summer School on Generative and Transformational Techniques in Software Engineering.* (Joao M. Fernandes and Ralf Lammel and Joao Saraiva and Joost Visser, Eds.), LNCS 6491, Springer. (2010)

[13] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur et Laurence Duchien. **Software Architecture Evolution**. dans *Software Evolution.* (Tom Mens and Serge Demeyer eds, Eds.), Springer Verlag. 2008. pages 233–262.

[14] Olivier Barais et Laurence Duchien. **SafArchie Studio : An ArgoUML extension to build Safe Architectures.** dans *Architecture Description Languages.* (en, Eds.), Springer, pages 85–100.

Conferences et Symposiums (proceedings) (Avec édition d'actes et comité de sélection)

[15] Guillaume Bécan, Nicolas Sannier, Mathieu Acher, Olivier Barais, Arnaud Blouin, Benoit Baudry **Automating the Formalization of Product Comparison Matrices**, In *Proceeding of the International Conference on Automated Software Engineering (ASE), 2014*. September 15 - 19, 2014 Västerås, Sweden.

[16] Thomas Hartmann, Francois Fouquet, Gregory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon **A Native Versioning Concept to Support Historized Models at Runtime**, In *ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*, Valencia, Spain, september 28 - october 03, ACM.

[17] Mathieu Acher, Benoit Baudry, Olivier Barais and Jean-Marc Jézéquel **Customization and 3D Printing : A Challenging Playground for Software Product Lines**, In *18th International Software Product Line Conference (2014)*, Firenze, Italy, September 15-19, 2014, ACM.

[18] Dimitri Van Landuyt, Steven Op De Beeck, Aram Hovsepyan, Sam Michiels, Wouter Joosen, Sven Meynckens, Gjalt De Jong, Olivier Barais, Mathieu Acher **Towards Managing Variability in the Safety Design of an Automotive Hall Effect Sensor**, In *18th International Software Product Line Conference (2014)*, Firenze, Italy, September 15-19, 2014, ACM.

[19] Julien Richard-Foy, Olivier Barais, Jean-Marc Jézéquel **Using Path-Dependent Types to Build Type Safe JavaScript Foreign Function Interfaces**. In *14th International Conference on Web Engineering (ICWE'2014)*, Toulouse, France, July 1-4 July.

[20] Inti Y. Gonzalez-Herrera, Johann Bourcier, Erwan Daubert, Walter Rudametkin, Olivier Barais, François Fouquet, Jean-Marc Jézéquel **Scapegoat : An Adaptive Monitoring Framework for Component-Based Systems**. In *2014 IEEE/IFIP Conference on Software Architecture, WICSA 2014*, Sydney, Australia, April 7-11, 2014, 67-76, IEEE.

[21] Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet, Jean-Marc Jézéquel **Improving Reusability in Software Process Lines**. In *39th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2013* , Santander, Spain, September 4-6, 2013, 90-93. IEEE.

[22] Julien Richard-Foy, Olivier Barais and Jean-Marc Jézéquel **Efficient High-Level Abstractions for Web Programming**. In *12th International Conference on Generative Programming : Concepts & Experiences (GPCE'13)*, October 27-28, 2013 Indianapolis, IN, USA

[23] Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, David Touzet and Jean-Marc Jézéquel. **Integrating Software Process Reuse and Automation**. In *The 20th Asia-Pacific Software Engineering Conference (APSEC 2013)* 2-5 December 2013, Bangkok, Thailand

[24] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry and Robert France. **Reifying Concurrency for Executable Metamodeling**. In *6th International Conference on Software Language Engineering (SLE 2013)*, Indianapolis, FL, USA, in Springer's Lecture Notes in Computer Science series.

[25] Mathieu Acher, Benoit Combemale, Philippe Collet, Olivier Barais, Philippe Lahire and Robert B. France. **Composing your Compositions of Variability Models**. In *ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MODELS'13)*, 2013, Miami, Florida - USA 29/09 - 4/10

[26] Joao Bosco Ferreira Filho, Olivier Barais, Mathieu Acher Jérôme Le Noir and Benoit Baudry. **Generating Counterexamples of Model-based Software Product Lines : An Exploratory Study**. In *17th International Conference on Software Product Lines (SPLC'13)*, 2013, Tokyo - Japan, August 26-30,

[27] Max E. Kramer, Jacques Klein, Jim R. H. Steel, Brice Morin, Jörg Kienzle, Olivier Barais, Jean-Marc Jézéquel : **Achieving Practical Genericity in Model Weaving through Extensibility**. *6 th International Conference on Model Transformation (ICMT'2013)* : 108-124, Budaptest, Hungary, June 18-19.

[28] Emmanuelle Rouillé, Benoît Combemale, Olivier Barais, Jean-Marc Jézéquel and David Touzet. **Leveraging CVL to Manage Variability in Software Process Lines**. *The 19th Asia-Pacific Software Engineering Conference (APSEC 2012)*. Hong Kong, December 04-07.

[29] François Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, Jean-Marc Jézéquel. **A dynamic component model for cyber physical systems**. *In the 15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE 2012)*. Bertinoro, Italy, June 26-28, 2012. 135-144

[30] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, Jean-Marc Jézéquel. **Dissemination of Reconfiguration Policies on Mesh Networks**. *In the 12th International IFIP Conference on Distributed Applications and Interoperable Systems (DAIS 2012)*. Stockholm, Sweden - June 13-16, 2012, 16-30

[31] Olivier Beaudoux, Mickael Clavreul, Arnaud Blouin, Mengqiang Yang, Olivier Barais, Jean-Marc Jézéquel. **Specifying and running rich graphical components with Loa.** *In the fourth ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2012)* : 169-178

[32] Gilles Perrouin, Brice Morin, Franck Chauvel, Franck Fleurey, Jacques Klein, Yves Le Traon, Olivier Barais, Jean-Marc Jézéquel. **Towards flexible evolution of dynamically adaptive systems**. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 1353-1356.

[33] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. 2012. **An eclipse modelling framework alternative to meet the models@runtime requirements**. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems (MODELS'12)*, Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Springer-Verlag, Berlin, Heidelberg, 87-101.

[34] Juan F. Ingles-Romero, Cristina Vicente-Chicote, Brice Morin, Olivier Barais. **Towards the Automatic Generation of Self-Adaptive Robotics Software : An Experience Report**. dans *20th IEEE International Workshops on Enabling Technologies : Infrastructures for Collaborative Enterprises, (WETICE 2011)*. Paris, France, 27-29 June 2011, Proceedings. IEEE Computer Society 2011, 79-86

[35] Olivier Beaudoux, Arnaud Blouin, Olivier Barais, Jean-Marc Jézéquel. **Specifying and implementing UI data bindings with active operations**. dans *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing System, 'EICS 2011)*, Pisa, Italy, June 13-16, 2011. ACM 2011 127-136.

[36] Franck Fleurey, Brice Morin, Arnor Solberg et Olivier Barais. **MDE to Manage Communications with and between Resource-Constrained Systems**. dans *MODELS'11 : ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, Wellington, New Zealand, October 2011.

[37] Brice Morin, Tejeddine Mouelhi, Franck Fleurey, Yves Le Traon, Olivier Barais et Jean-Marc Jézéquel. **Security-Driven Model-Based Dynamic Adaptation**. dans *25nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*. Antwerp, Belgium, September. (2010)

[38] Francoise André, Erwan Daubert, Grégory Nain, Brice Morin, Brice et Olivier Barais. **F4Plan : An Approach to build Efficient Adaptation Plans**. dans *MobiQuitous 2010*. (2010)

[39] Grégory Nain, François Fouquet, Brice Morin, Olivier Barais et Jean-Marc Jézéquel. **Integrating IoT and IoS with a Component-Based approach**. dans *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*. Lille, France. (2010)

[40] Mickael Clavreul, Olivier Barais et Jean-Marc Jézéquel. **Integrating Legacy Systems with MDE**. dans *ICSE'10 : Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. Cape Town, South Africa, May, pages 69–78. (2010)

[41] Olivier Beaudoux, Arnaud Blouin, Olivier Barais et Jean-Marc Jézéquel. **Active Operations on Collections**. dans *ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS'10)*. Oslo, Norway, pages 91-105. (2010)

[42] Naouel Moha, Vincent Mahé, Olivier Barais et Jean-Marc Jézéquel. **Generic Model Refactorings.** dans *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*. Denver, Colorado, USA, Oct 2009.

[43] Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt et Jean-Marc Jézéquel. **Weaving Variability into Domain Metamodels.** dans *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*. Denver, Colorado, USA, Oct 2009.

[44] Brice Morin, Thomas Ledoux, Mahmoud Ben Hassine, Franck Chauvel, Olivier Barais et Jean-Marc Jézéquel. **Unifying Runtime Adaptation and Design Evolution.** dans *IEEE 9th International Conference on Computer and Information Technology* (CIT'09). Xiamen, China, Oct 2009.

[45] Brice Morin, Olivier Barais, Grégory Nain et Jean-Marc Jézéquel **Taming Dynamically Adaptive Systems with Models and Aspects**. dans *31st International Conference on Software Engineering (ICSE'09)*, Mai 2009, Vancouver, Canada.acceptation rate : 12%

[46] Grégory Nain , Olivier Barais, Régis Fleurquin et Jean-Marc Jézéquel. **EntiMid : un middleware aux services de la maison.** dans *3ème Conférence Francophone sur les Architectures Logicielles*, Nancy, France, Mars 2009.

[47] Grégory Nain , Erwan Daubert , Olivier Barais et Jean-Marc Jézéquel. **Using MDE to Build a Schizonfrenic Middleware for Home/Building Automation.** dans *ServiceWave'08 : Networked European Software & Services Initiative (NESSI) Conference*. Madrid, Spain, Décembre 2008. acceptation rate : 26%

[48] Franck Chauvel , Olivier Barais , Isabelle Borne et Jean-Marc Jézéquel. **Composition of Qualitative Adaptation Policies.** dans *23rd IEEE/ACM International Conference on Automated Software Engineering - ASE'08*. L'Aquila, Italy, Septembre 2008. acceptation rate : 27%

[49] Brice Morin , Gilles Vanwormhoudt , Philippe Lahire , Alban Gaignard , Olivier Barais et Jean-Marc Jézéquel. **Managing Variability Complexity in Aspect-Oriented Modeling**. dans *MoDELS'08 : 11th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Toulouse, France, Octobre 2008. acceptation rate : 21%

[50] Freddy Munoz , Benoit Baudry et Olivier Barais. **Improving Maintenance in AOP Through an Interaction Specification Framework.** dans *24th IEEE International Conference onSoftware Maintenance - ICSM'08*. Beijing, China, Septembre 2008. acceptation rate : 24%

[51] Franck Chauvel , Olivier Barais , Noel Plouzeau , Isabelle Borne et Jean-Marc Jézéquel **Expression qualitative de politiques d'adaptation pour Fractal.** dans *Langage Modèles et Objets LMO'08*. Montréal, Quebec, Mars 2008. acceptation rate : 30%

[52] Diego Alonso , Cristina Vicente-Chicote et Olivier Barais **V3Studio : A Component-Based Architecture Modeling Language.** dans *15th IEEE International Conference on Engineering of Computer-Based Systems (ECBS'O8)*. Belfast, Northern Ireland, Avril 2008.

[53] Olivier Barais , Jacques Klein , Benoit Baudry , Andrew Jackson et Siobhan Clarke. **Composing Multi-View Aspect Models.** dans *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*. Madrid, Spain, Février 2008. acceptation rate : 39%

[54] Franck Chauvel , Isabelle Borne , Jean-Marc Jézéquel et Olivier Barais. **A Model-Driven Process for Self-Adaptive Software.** dans *4th European Congress ERTS Embedded Real-Time Software*. Toulouse, France, Janvier 2008.

[55] Sébastien Saudrais, Olivier Barais, et Noël Plouzeau. **Integration of time issues into component-based applications**. dans *the 10th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE'07)*. Springer Lecture Notes in Computer Science (LNCS), Juillet 2007, pages : 173–188. acceptation rate : 21%

[56] Rodrigo Ramos, Olivier Barais et Jean-Marc Jézéquel. **Matching Model-Snippets**. dans *Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*. Nashville, TN, USA, Octobre 2007 ;pages 121–135 acceptation rate : 26% **selected as one of the best papers**

[57] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais et Jean-Marc Jézéquel. **Introducing variability into Aspect-Oriented Modeling approaches.** dans *Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*. Nashville, TN, USA, Octobre 2007, pages : 498–513. acceptation rate : 26%

[58] Jacques Klein, Benoit Baudry, Olivier Barais et Andrew Jackson. **Introduction du test dans la modélisation par aspects**. dans *Troisième Journées sur l'Ingénierie Dirigée par les Modèles (IDM'2007)*. Toulouse, France, Mars 2007.

[59] Sébastien Saudrais, Olivier Barais, Laurence Duchien et Noël Plouzeau **Intégration de propriétés temporelles dans des applications à base de composants**. dans *Dixième Anniversaire de la Conférence Francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'07)*. Namur, Belgium, Juin 2007.

[60] Sébastien Saudrais, Olivier Barais et Noël Plouzeau. **Composants avec Propriétés Temporelles**. dans *Proceedings of the Conférence sur les Architecture Logicielle (CAL'2006)*. Nantes, France, Septembre 2006.

[61] Olivier Barais, Julia Lawall, Julia, Anne-Françoise Le Meur et Laurence Duchien. **Safe Integration of New Concerns in a Software Architecture.** dans *Proceedings of the 13th International Conference on Engineering of Computer Based Systems (ECBS'06)*. Potsdam, Germany, Mars 2006. IEEE, pages 52-64.

[62] Olivier Barais, Julia Lawall, Anne-Françoise Le Meur et Laurence Duchien. **Providing Support for Safe Software Architecture Transformations.** dans *Working Session of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005)*. Pittsburg, PA, USA, Novembre 2005.

[63] Olivier Barais, Laurence Duchien et Anne-Francoise Le Meur. **A Framework to Specify Incremental Software Architecture Transformations.** dans *31st EUROMICRO CONFERENCE on Software Engineering and Advanced Applications (SEAA 2005)*. Septembre 2005. IEEE Computer Society, pages 62–70.

[64] Olivier Barais et Laurence Duchien. **SafArch : Maîtriser l'Evolution d'une Architecture Logicielle.** Dans *Langages, Modèles & Objets, Journées Composants, LMO 2004-JC 2004*. Lille, France, Mars 2004. Hermès Sciences, pages 103–116.

Workshop (proceedings) (Avec édition d'actes et comité de sélection)

[65] Marianela Ciolfi Felice, Joao Bosco Filho Ferreira, Mathieu Acher, Arnaud Blouin and Olivier Barais. **Interactive Visualisation of Products in Online Configurators : A Case Study for Variability Modelling Technologies**. In *MAPLE/SCALE 2013 at SPLC 2013 Joint Workshop of MAPLE 2013 – 5th International Workshop on Model-driven Approaches in Software Product Line Engineering and SCALE 2013 – 4th Workshop on Scalable Modeling Techniques for Software Product Lines*, 2013, Tokyo - Japan, August 26-30.

[66] João Bosco Ferreira Filho, Olivier Barais, Jérôme Le Noir and Jean-Marc Jézéquel. **Customizing the Common Variability Language Semantics for your Domain Models**. In *Vary'12 Workshop@Models'12*. Springer-Verlag, Berlin, Heidelberg.

[67] Benoit Combemale, Olivier Barais, Omar Alam and Jörg Kienzle. **Using CVL to Operationalize Product Line Development with Reusable Aspect Models**. In *Vary'12 Workshop@Models'12*. Springer-Verlag, Berlin, Heidelberg.

[68] João Bosco Ferreira Filho, Olivier Barais, Benoit Baudry, Windson Viana, and Rossana M. C. Andrade. 2012. **An approach for semantic enrichment of software product lines**. In *Proceedings of the 16th International Software Product Line Conference* - Volume 2 (SPLC '12), Vol. 2. ACM, New York, NY, USA, 188-195.

[69] Juan F. Inglés-Romero, Cristina Vicente-Chicote, Brice Morin et Olivier Barais. **Using ModelsRuntime for Designing Adaptive Robotics Software : an Experience Report**. Dans *1st international workshop on Model Based Engineering for Robotics : RoSym'10 at (MODELS'10)*. Oslo, Norway. (2010)

[70] Brice Morin, Franck Fleurey, Olivier Barais et Jean-Marc Jézéquel. **Aspect-Oriented Modeling to Support Dynamic Adaptation**. dans *Forum Demo at AOSD'10*. Rennes and St Malo, France. (2010)

[71] Marie Gouyette, Olivier Barais, Jérôme Le Noir et Jean-Marc Jézéquel. **Managing variability in multi-views engineering : A live demo**. dans *Journée Lignes de Produits*. Paris, France, October. (2010)

[72] François Fouquet, Olivier Barais et Jean-Marc Jézéquel. **Building a Kermeta Compiler using Scala : an Experience Report**. dans *Scala Days 2010*. Lausanne, Switzerland. EPFL. (2010)

[73] Brice Morin, Grégory Nain, Olivier Barais et Jean-Marc Jézéquel **Leveraging Models From Design-time to Runtime. A Live Demo**. dans *4th International Workshop on Models@Run.Time (at MODELS'09)*. Denver, Colorado, USA, Oct 2009.

[74] Franck Chauvel, Olivier Barais, Noël Plouzeau, Isabelle Borne, et Jean-Marc Jézéquel. **Composition et expression qualitative de politiques d'adaptation pour les composants fractal.** Dans *Actes des Journées nationales du GDR GPL 2009*, Toulouse, France, January 2009

[75] Brice Morin , Olivier Barais et Jean-Marc Jézéquel. **K@RT : An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines.** dans *Proceedings of the Models Workshop on Mo-*

dels@Runtime'08. Toulouse, France, October 2008.

[76] Brice Morin , Olivier Barais et Jean-Marc Jézéquel. **Weaving Aspect Configurations for Managing System Variability.** dans *In Second International Workshop on Variability Modelling of Software-intensive Systems.* Essen, Germany, Janvier 2008

[77] Brice Morin , Jacques Klein , Olivier Barais et Jean-Marc Jézéquel. **A Generic Weaver for Supporting Product Lines**. dans *International Workshop on Early Aspects at ICSE'08.* Leipzig, Germany, Mai 2008.

[78] Sébastien Saudrais, Olivier Barais et Noel Plouzeau. **Monitoring your Lego Mindstorms with Giotto**. dans *Proceedings of ARTIST International Workshop on Tool Platforms for Modeling, Analysis and Validation of Embedded Systems.* Berlin, Germany, Juillet 2007.

[79] Brice Morin, Olivier Barais, Jean-Marc Jézéquel et Rodrigo Ramos. **Towards a generic aspect-oriented modeling framework**. dans *Models and Aspects workshop, at ECOOP 2007*, July 2007.

[80] Pierre-Alain Muller et Olivier Barais. **Control-theory and models at runtime.** dans *Proceedings of the Models Workshop on ModelsRuntime.* Nashville, USA, Octobre 2007.

[81] Freddy Munoz, Olivier Barais et Benoit Baudry. **Vigilant usage of Aspects**. dans *Proceedings of ADI 2007 - Workshop on Aspects, Dependencies, and Interactions at ECOOP 2007.* Berlin, Germany, Juillet 2007.

[82] Jean-Marie Mottu, Olivier Barais, Mark Skipper, Didier Vojtisek et Jean-Marc Jézéquel. **Intégration du support OCL dans Kermeta. Spécifiez la sémantique statique de vos méta-modèles.** dans *Dixième Anniversaire de la Conférence Francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'07).* Namur, Belgique, 2007.

[83] Olivier Barais. **Séparation des préoccupations en phase de méta-modélisation**. dans *Atelier Composition de modèles (Cosmo) de la Conférence Langage et Modèles à Objets (LMO'07) et de la 3 ème Journée sur l'Ingénierie Dirigée par les Modèles.* Toulouse, France 2007.

[84] Olivier Barais, Franck Fleurey, Pierre-Alain Muller, Didier Vojtisek et Jean-Marc Jézéquel. **Nouvelles fonctionnalités de Kermeta.** dans *Session Démonstrations des 3ème Journées sur l'Ingénierie Dirigée par les Modèles.* Toulouse, France 2007.

[85] Olivier Barais. **SpoonEMF, une brique logicielle pour l'utilisation de l'IDM dans le cadre de la réingénierie de programmes Java5**. dans *2ème Journée sur l'Ingénierie Dirigée par les Modèles.* Lille, France 2006.

[86] Andrew Jackson, Olivier Barais, Jean-Marc Jézéquel et Siobhán Clarke. **Toward A Generic And Extensible Merge Operator.** dans *Models and Aspects workshop, at ECOOP 2006.* Nantes, France, Juillet 2006.

[87] Sébastien Saudrais, Olivier Barais et Laurence Duchien. **Using Model-Driven Engineering to generate QoS Monitors from a formal specification.** dans *Proceedings of the Aquserm 2006.* Hong Kong, China, Octobre 2006.

[88] Han-Missi Tran, Olivier Barais, Anne-Françoise Le Meur et Laurence Duchien. **Safe Integration of New Concerns in a Software Architecture : Overview of the Implementation.** dans *the 2nd International ECOOP Workshop on Architecture Centric Evolution (ACE'06).* Nantes, France, Juillet 2006.

[89] Nicolas Pessemier, Olivier Barais, Lionel Seinturier, Thierry Coupaye et Laurence Duchien. **A Three Level Framework for Adapting Component Based Architectures**. dans *2nd Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT) at ECOOP'05.* Juillet 2005.

[90] Olivier Barais, Alexis Muller et Nicolas Pessemier. **Extension de Fractal pour le Support des Vues au sein d'une Architecture Logicielle.** Dans *Objets Composants et Modèles dans l'ingénierie des SI* (OCM-SI 2004). Biarritz, France, Juin 2004.

[91] Nicolas Pessemier, Lionel Seinturier, Laurence Duchien et Olivier Barais. **Une extension de Fractal pour l'AOP.** Dans *Première journée Francophone sur le Développement de Logiciels par Aspects* (JFDLPA 2004). Paris, France, septembre 2004.

[92] Olivier Barais et Laurence Duchien. **SafArchie Studio : An ArgoUML extension to build Safe Architectures.** Dans *Workshop on Architecture Description Languages* (WADL 2004). Toulouse, France, Août 2004.

[93] Olivier Barais, Eric Cariou, Laurence Duchien, Nicolas Pessemier et Lionel Seinturier. *TranSAT : A Framework for the specification of Software Architecture Evolution.* Dans *ECOOP First International Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT04).* Oslo, Norway, Juin 2004.

[94] Olivier Barais, Laurence Duchien et Renaud Pawlak. **Separation of Concerns in Software Modeling : A Framework for Software Architecture Transformation.** Dans *IASTED International Conference on Software Engineering Applications*, SEA 2003. Los Angeles, USA, Novembre. ACTA Press, pages 663–668.

Tutorial

[95] Francois Fouquet, Erwan Daubert, Grégory Nain, Brice Morin, Johann Bourcier, and Olivier Barais. **Designing and Evolving Distributed Architecture using Kevoree.** In *17h International ACM SIGSOFT Symposium on Component-Based Software Engineering*, Lille, France, June 30 to July 4, 2014.

[96] Francois Fouquet, Erwan Daubert, Grégory Nain, Brice Morin, Johann Bourcier, and Olivier Barais. **Designing and Evolving Distributed Architecture using Kevoree.** In *ACM/IFIP/USENIX International Middleware Conference*, 2013 Bejing China, Decembre 9-13.

[97] Mathieu Acher, Benoit Combemale and Olivier Barais. **Model-Based Variability Management**. In *ECMFA, ECOOP and ECSA'13.* (2013) Montpellier, France July 1-2.

[98] Jérôme Le Noir, Olivier Barais, João Bosco Ferreira Filho, Jean-Marc Jézéquel et Mathieu Acher. **Modelling variability using CVL ; A step by step tutorial.** Dans *la Journée Lignes de Produits*, le 6 novembre 2012 à Lille.

[99] Marie Gouyette, Olivier Barais, Jérôme, Lenoir, Cédric Brun, Marcos Almeida Da Silva, Xavier Blanc and Jean-Marc Jezequel. **Movida Studio : a modeling environment to create viewpoints and manage variability in views.** Dans *CAL, IDM, COSMAL et GDR GPL'11*, Lille, France 2011.

Thèse

[100] Olivier Barais. **Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants**. *PhD thesis, Laboratoire d'Informatique Fondamentale de Lille/ Université de Lille 1.* Novembre 2005.

Récapitulatif des publications pour les années universitaires 2002/2014

RECAPITULATIF	International(e)	National(e)	Total
Revue	7	4	11
Chapitre d'ouvrage	3	0	3
Conférence	44	6	50
Atelier	24	6	30
Rapport de recherche	2	4	6
Total	81	19	100

---

10 Mobilité thématique et/ou géographique

Fin 2005, je suis parti en post-doctorat au sein de l'équipe Triskell à Rennes. À l'issue de ce post-doctorat, j'ai eu l'opportunité d'intégrer l'équipe des enseignants-chercheurs de l'Institut de Formation Supérieure en Informatique et Communication (IFSIC) de l'université de Rennes 1. Malgré cette triple évolution et une première année d'enseignement chargée ($\succ$ 250 heures), l'année 2007 fut une année riche pour mon activité de recherche (une dizaine de publications dont deux journaux, un chapitre de livre et trois conférences internationales de très haut niveau en génie logiciel). Cette mobilité géographique a entraîné aussi une mobilité thématique : d'une thèse sur les architectures logicielles, j'ai intégré une équipe dont la thématique de recherche principale est l'ingénierie dirigé par les modèles. Cette mobilité géographique et thématique me permet maintenant de travailler avec quatre communautés de recherche du domaine du génie logiciel :

— la communauté : *Component Based Software Engineering*,
— la communauté : *Model Driven Engineering*,
— la communauté : *Aspect Oriented Modelling*.
— la communauté : *Software Product Line Engineering*.

Résumé

Le développement logiciel traditionnel, généralement fondé sur l'hypothèse d'un monde clos définissant une frontière connue et stable entre le système et son environnement n'est plus tenable. Par opposition, la notion de système dit ouvert et éternel s'est imposée à la plupart des systèmes informatiques. Ces systèmes logiciels se caractérisent par leur besoin d'offrir des capacités d'adaptation qui leur permettent de réagir aux changements de leur environnement de manière continue et sans interruption de service.

Un des challenges important pour la communauté du génie logiciel est d'identifier et de supprimer progressivement les limites liées à l'hypothèse du monde clos. En partant de cette hypothèse de monde ouvert, cette habilitation expose les bénéfices engendrés par l'effacement de la frontière entre la phase de conception et la phase d'exécution du logiciel en proposant l'utilisation des travaux liés à la modélisation non plus uniquement lors de la phase de conception du système, mais aussi au cours de l'exécution des systèmes dits ouverts.

Pour ce faire, cette habilitation synthétise, dans une première partie, les fondations d'une approche permettant l'utilisation de techniques de modélisation, à l'exécution en se concentrant principalement sur le point de vue de l'architecte logicielle. Nous exposons ensuite les bénéfices attendus en montrant comment des approches avancées de composition logicielle, de vérification ou de gestion de la variabilité peuvent être bénéfiques pour la compréhension et la maîtrise de l'espace de configuration et de reconfiguration d'un système dit ouvert. Nous synthétisons ensuite les principaux challenges liés à l'utilisation de techniques de modélisation à l'exécution en particulier dans le cadre de systèmes distribués et hétérogènes. Afin d'insister dans ce document de synthèse sur l'importance de l'expérimentation dans ma démarche scientifique, une deuxième partie présente un résumé non exhaustif, mais représentatif, de différentes expériences menées dans le cadre des thèses que j'encadre ou que j'ai co-encadrées afin de montrer la pertinence des approches de modélisation à l'exécution et des opérateurs de composition de modèles associés. Ces expériences permettent de confronter cette proposition à différents domaines d'applications : informatique mobile, Internet des Objets, Cloud Computing) et permettent de pousser aux limites cette idée d'utilisation de modélisation à l'exécution en regardant sa pertinence pour chaque domaine étudié par rapport à ses propres contraintes. Une dernière partie conclut ce manuscrit de synthèse par quelques chiffres pour étayer la production scientifique et propose un ensemble de perspectives de recherche associées à ces travaux.