

**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 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 Pazzat, 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

<b>Table des matières</b>	<b>2</b>
<b>1 Introduction</b>	<b>7</b>
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
<b>I Vers une convergence de l'espace de conception et de l'espace d'exécution</b>	<b>15</b>
<b>2 Principe du Models@runtime</b>	<b>19</b>
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
<b>3 Améliorations des techniques de modélisation pour une utilisation à l'exécution</b>	<b>27</b>
3.1 Vers une diminution du couplage par rapport au métamodèle . . . . .	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
<b>4</b>	<b>Modèle à l'exécution pour la gestion de systèmes adaptatifs hétérogènes et distribués</b>	<b>43</b>
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
<b>II</b>	<b>Expérimentations dans le cadre de ces recherches</b>	<b>55</b>
<b>5</b>	<b>Models@Runtime en environnement mobile</b>	<b>59</b>
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 <i>gossip</i> .	61
5.1.3	Principe de combinaison des horloges vectorielles ainsi qu'une propagation <i>gossip</i> . . . . .	62
5.1.4	Protocole <i>gossip</i> pour dissémination de <i>Model@Runtime</i> . . . . .	62
5.1.4.1	Algorithme principal (voir partie algorithme 3) . . . . .	63
5.1.4.2	Fonction <i>SelectPeer</i> (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	<i>Slicing</i> de modèle à l'image du <i>peer sampling</i> . . . . .	67
5.2	Validation expérimentale sur <i>cluster</i> 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
<b>6</b>	<b>Models@runtime pour le cloud computing</b>	<b>79</b>
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
<b>7</b>	<b>Models@runtime pour les objets connectés</b>	<b>103</b>
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	Résultats et analyse expérimentale . . . . .	110
7.5.4	Extension expérimentale pour connaître 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
<b>8</b>	<b>Models@runtime pour les systèmes de surveillance dynamiques et optimistes</b>	<b>117</b>
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
<b>9</b>	<b>Kevoree : une synthèse des idées sous la forme d'un projet <i>open source</i></b>	<b>131</b>
9.1	Outilage 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
<b>III Conclusion et perspectives</b>	<b>143</b>
<b>10 Bilans d'activités</b>	<b>145</b>
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
<b>11 Perspectives et Projet de Recherche</b>	<b>149</b>
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
<b>Bibliographie</b>	<b>159</b>
<b>Table des figures</b>	<b>173</b>
<b>IV Sélection d'articles scientifiques</b>	<b>175</b>
<b>V CV détaillé</b>	<b>287</b>

# 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*<sup>1</sup> 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 importants 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<sup>+</sup>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<sup>+</sup>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 prédictats 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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 où 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.

À 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'opérationnalisation 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 entre 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 métamodélisation, 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<sup>+</sup>09], présente cette vision générale.

Le chapitre 3 résume et commente quatre contributions [SMM<sup>+</sup>12, CBJ10, FFBA<sup>+</sup>14, FNM<sup>+</sup>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<sup>+</sup>12b, FMF<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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

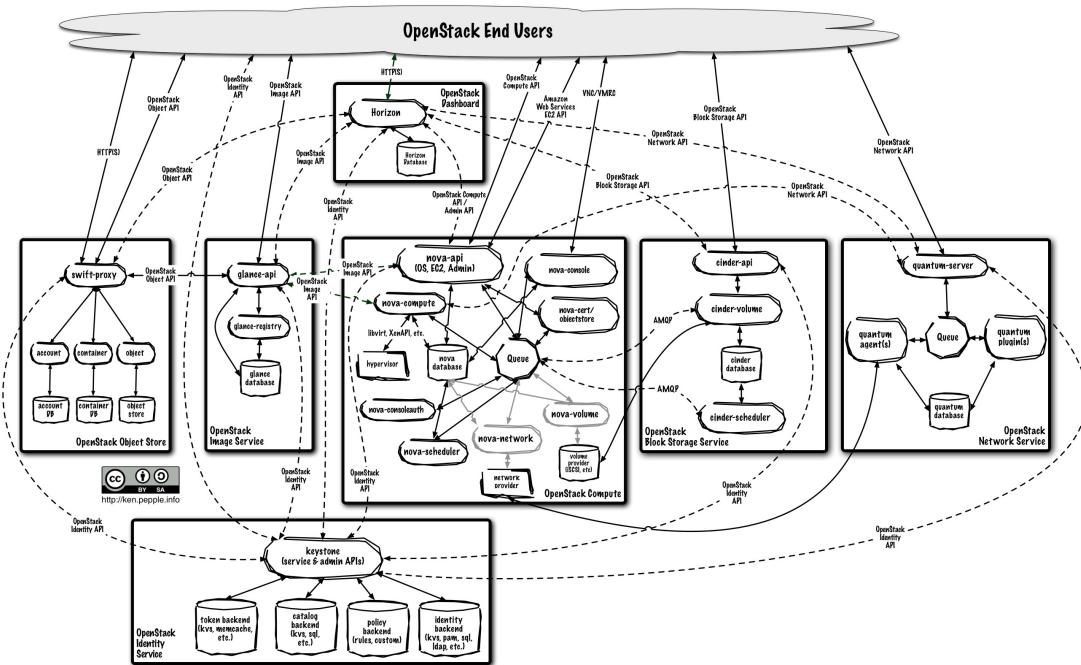


FIGURE 2.1 – Openstack architeture from [Opeb]

niveaux :

- tout d'abord, au travers d'une analyse du projet devstack<sup>1</sup> 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<sup>+</sup>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<sup>+</sup>98] que l'on peut voir comme un langage qui a indéniablement influencé l'initiative autour de Fractal [BCL<sup>+</sup>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*<sup>2</sup>. 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<sup>+</sup>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 métamodèles/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<sup>+</sup>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 adaptable 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<sup>+</sup>09].

### 2.2.2 Modélisation par Aspects pour la Dérivation de Configurations

La communauté SPL [BGH<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 *Trans-SAT* [LMV<sup>+</sup>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].

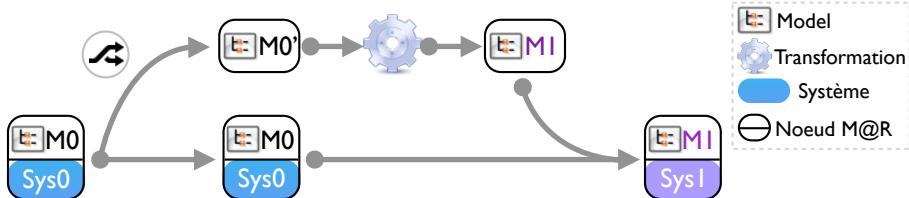
### 2.2.3 (Dé)coupler le modèle de réflexion de la réalité

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<sup>+</sup>12],[RBD<sup>+</sup>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 modélisé 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<sup>3</sup>. 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<sup>+12</sup>, CBJ10, FFBA<sup>+14</sup>, FNM<sup>+12</sup>]

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 systèmes 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<sup>+12</sup>, CBJ10, FFBA<sup>+14</sup>, FNM<sup>+12</sup>]

### 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 . . . . .	<b>33</b>
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 . . . . .	<b>37</b>
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 métta*<sup>1</sup> 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<sup>+</sup>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<sup>+</sup>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'expliquer 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

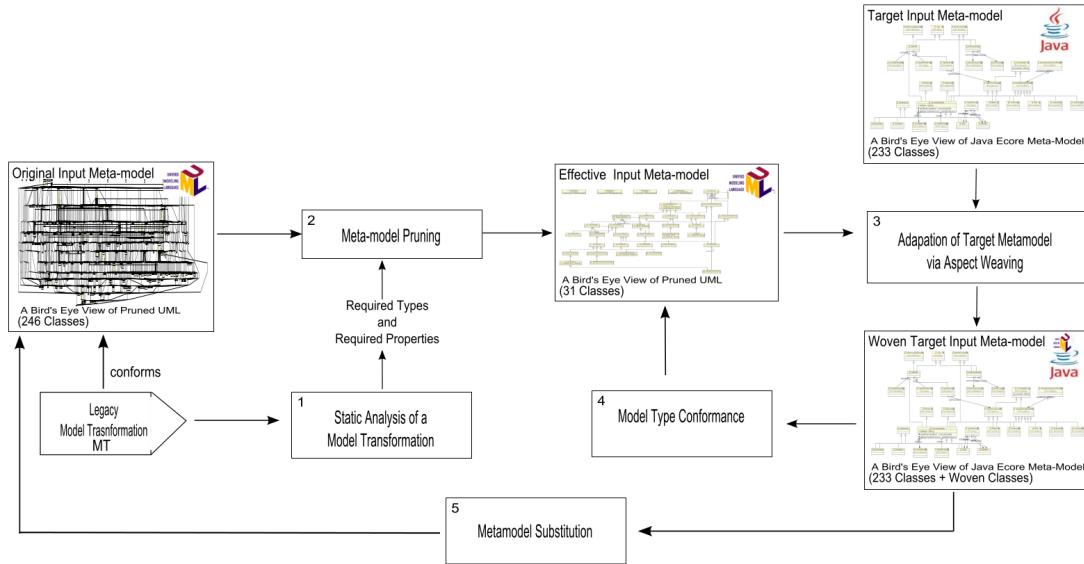


FIGURE 3.1 – Vue générale de l'approche

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 patrimonial differents 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.

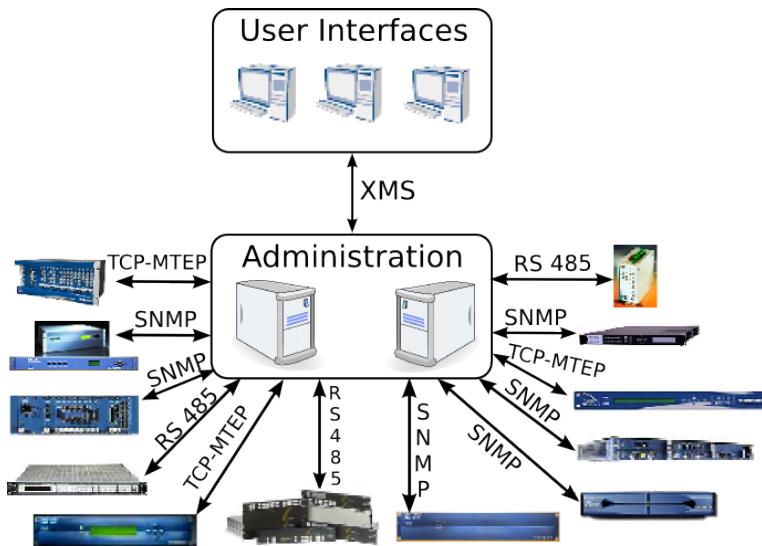


FIGURE 3.2 – Exemple de cas d'applications

Le langage dédié d'expression de correspondance s'appelle ModMap<sup>2</sup>. 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<sup>+14</sup>]

#### 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<sup>+12</sup>]. 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, entre 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 croît 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

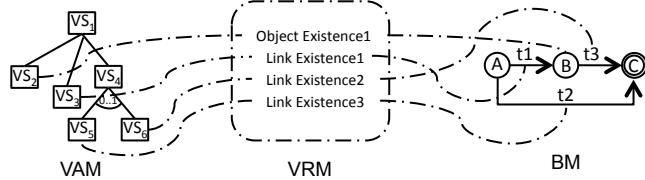


FIGURE 3.3 – Exemple de modèle CVL appliquée à un modèle de domaine de machine à états finie

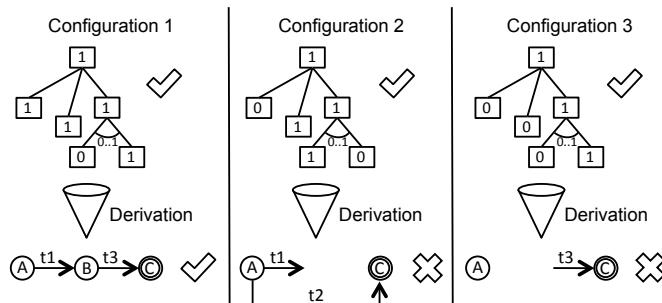


FIGURE 3.4 – Illustration schématique du problème

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  $C_1, C_2, C_3$ .

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 est de trouver, à partir d'un domaine métier défini par son métamodèle 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.

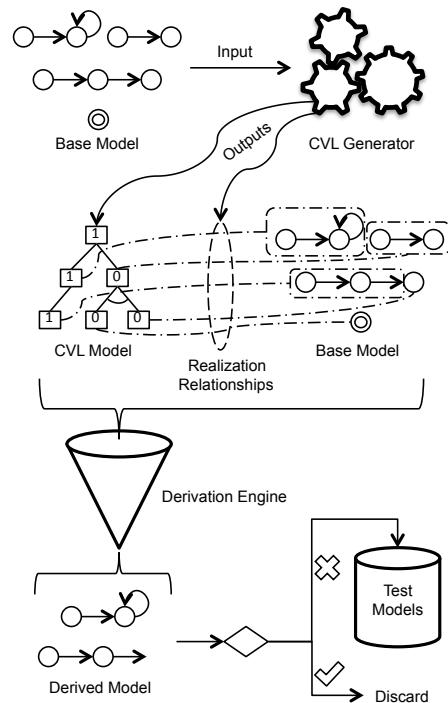


FIGURE 3.5 – Vue générale du processus automatisé pour la création de contre-exemples

### 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 conformité 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 ?

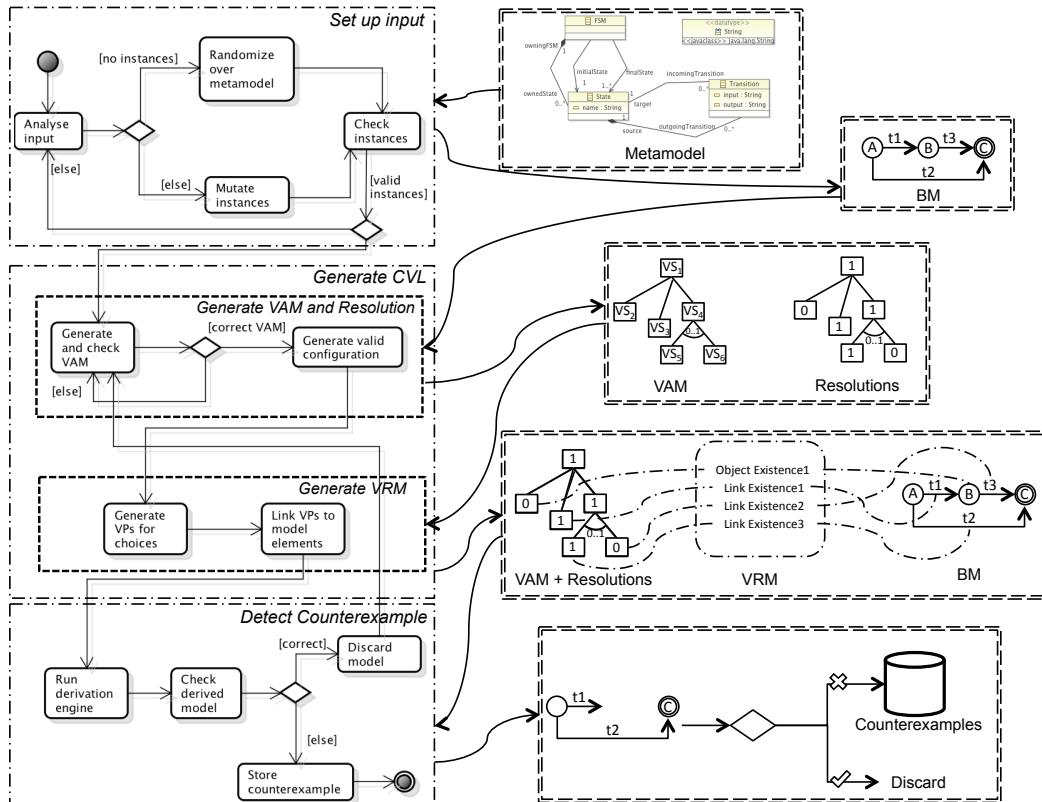


FIGURE 3.6 – Vue détaillée du processus automatisé pour la création de contre-exemples

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 métamodèle 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 générions 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 inconsistante 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 métamodèle. 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 métamodèle 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<sup>+12</sup>]

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 résultats 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<sup>+11</sup>] (mise en œuvre 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 élucider 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 élucidation 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 Élucidation 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<sup>+</sup>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 entre 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 finis 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 finis de quatre métaclasses (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

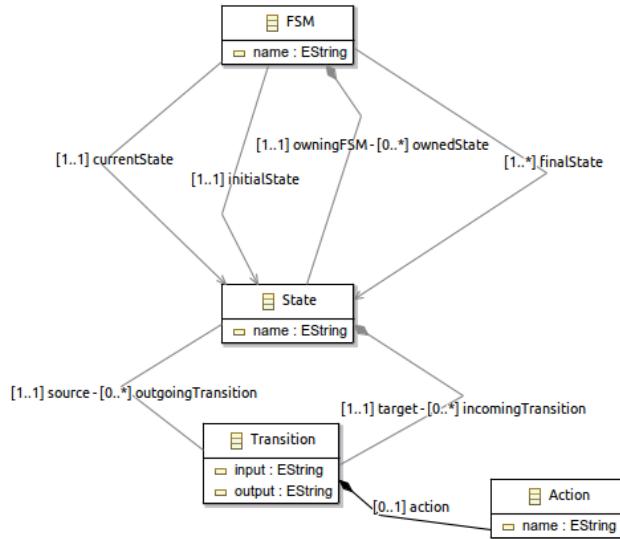


FIGURE 3.7 – Méta-modèle de machine à états finie utilisé dans le cadre de ces expériences

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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+12b</sup>, FMF<sup>+12</sup>]

### 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 dus 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<sup>1</sup>. 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 décorrérer 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 noeuds 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 noeuds 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 noeuds notamment dans le cadre de réseaux sporadiques, c'est-à-dire de réseaux sujet à de fréquentes erreurs et/ou de fréquentes déconnexions. 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 noeuds, 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*<sup>2</sup>. 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).

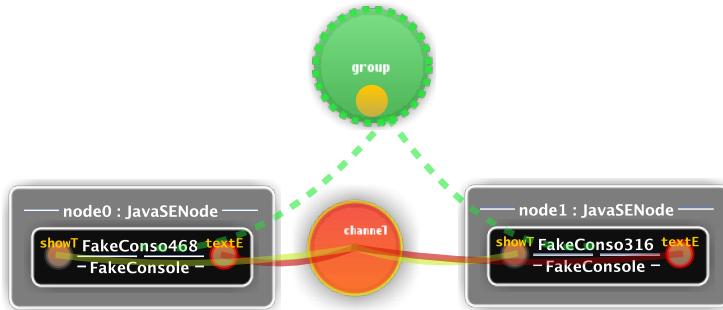


FIGURE 4.1 – Exemple d'application Kevoree

#### 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<sup>+</sup>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 noeuds. 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 boîtes 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, linstanciation d'un composant sur un noeud Java n'est pas la même que sur un noeud 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<sup>+</sup>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.

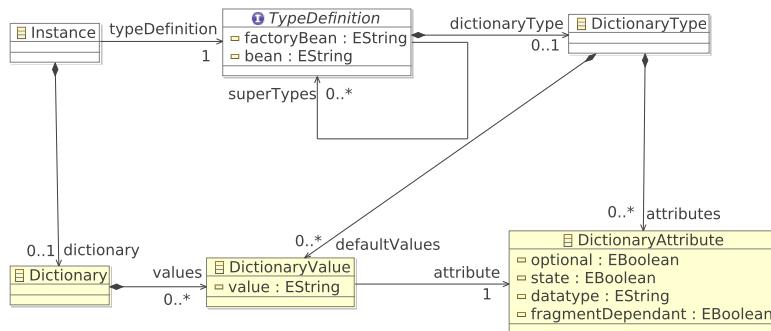


FIGURE 4.2 – Paradigme Type/Instance, dictionnaire et héritage de type

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, noeuds 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 noeuds distants. Enfin, les noeuds 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.

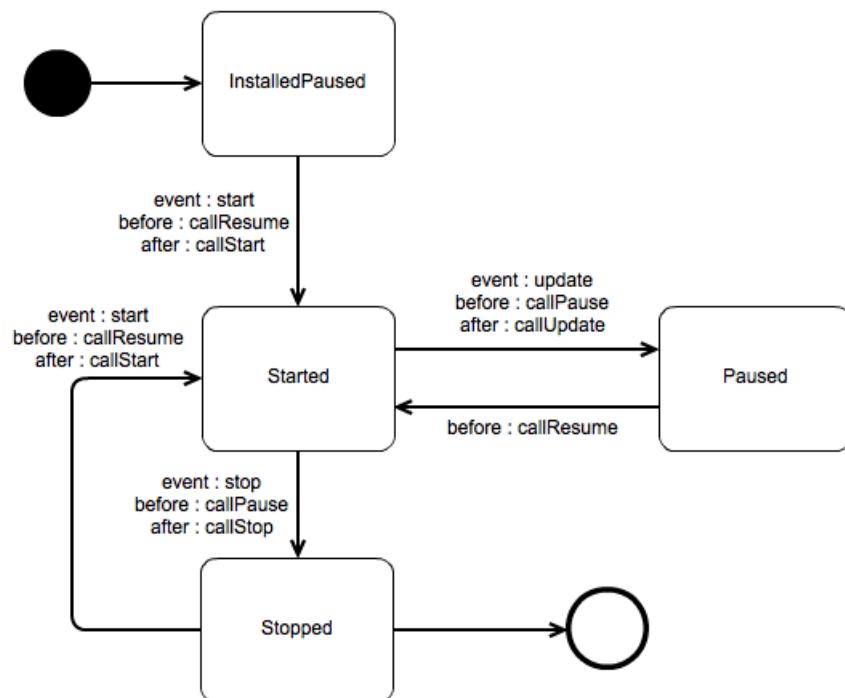


FIGURE 4.3 – Cycle de vie d'une instance

**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 *ModelListeners* d'être notifiés qu'une mise à jour a été proposée. Chaque *ModelListener* peut ainsi valider la configuration proposée.
- *preAllUpdate* permet aux *ModelListeners* 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 *ModelListeners* d'être notifiés que la mise à jour qui a été appliquée a aussi été validée par l'ensemble des *ModelListeners*.
- *preRollback* permet aux *ModelListeners* d'être notifiés que la mise à jour a échoué et qu'un retour à la configuration précédente va être effectué.
- *postRollback* permet aux *ModelListeners* 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 noeud 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 planter 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

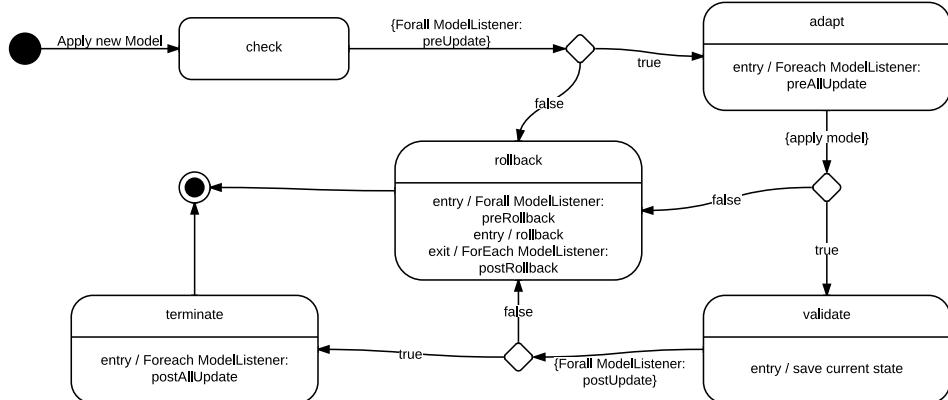


FIGURE 4.4 – Diagramme d’états-transitions montrant l’intégration des ModelListeners dans le processus de Model@Runtime

*ModelListener* sont utilisées pour la synchronisation et la validation des reconfigurations entre plusieurs noeuds. En effet, chaque noeud héberge un module Kevoree Core lui permettant d’évoluer indépendamment des autres noeuds. Mais les éléments présents sur chacun des noeuds sont capables de modifier l’ensemble du système et si la reconfiguration impacte plusieurs noeuds, il peut être nécessaire que plusieurs noeuds 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<sup>3</sup> 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 noeuds 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 autonomes [LMD13]. En effet, les nœuds sont les dépositaires de deux des phases de la boucle autonome [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).

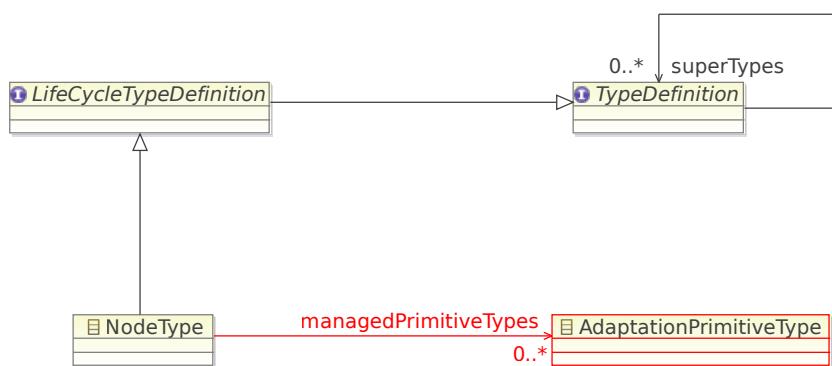


FIGURE 4.5 – Méta-modèle Kevoree avec la représentation des primitives d'adaptation

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 ordonner 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*.

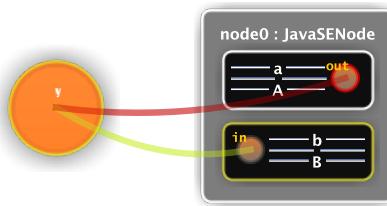


FIGURE 4.6 – Exemple de configuration nécessitant une planification

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 noeud, 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 noeud. C'est pourquoi le processus du Kevoree Core, qui délègue la comparaison et l'exécution au noeud, délègue aussi l'ordonnancement.

La définition d'un type de noeud 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<sup>4</sup>) (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 appliquée 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 entre 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 <i>gossip</i> . . . . .	61
5.1.3	Principe de combinaison des horloges vectorielles ainsi qu'une propagation <i>gossip</i> . . . . .	62
5.1.4	Protocole <i>gossip</i> pour dissémination de <i>Model@Runtime</i> . . . . .	62
5.1.5	Propriétés attendues . . . . .	65
5.1.6	<i>Slicing</i> de modèle à l'image du <i>peer sampling</i> . . . . .	67
5.2	Validation expérimentale sur <i>cluster</i> 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 importants et originaux 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<sup>1</sup> 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 noeud 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 noeuds 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 noeuds 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 noeud a besoin d'une valeur agrégée du *cluster* pour prendre des décisions mais l'appel à tous les noeuds est trop coûteux. A l'inverse, chaque noeud 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*<sup>2</sup>.

---

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-information 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<sup>+</sup>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 noeuds *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 noeud traversé par le message. L'horloge vectorielle sert alors à la traçabilité du message et surtout à sa synchronisation sur chaque noeud. 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<sup>3</sup> 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 noeud à une horloge. Le *nodeID* identifie ici le fragment du groupe courant déployé sur le noeud 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 noeud 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 noeuds 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 noeuds 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 dissème (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(neighbor, g)) :
8: result ← compareWithLocalVectorClock (remoteVectorClock)
9: if result == BEFORE || result == CONCURRENTLY then
10:   send (neighbor, ASK_MODEL)
11: end if
On receive (neighbor ∈ 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<sup>+07</sup>]. Différentes implantations de ce service sont disponibles dans la littérature avec différentes propriétés de convergence. Celle proposée ici est donc bien évidemment 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 noeuds 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 noeud, son score augmente de deux fois son taux d'échec précédent. Le score d'un noeud est remis au minimum des scores des noeuds lorsqu'une communication est établie après un échec. Le score d'un noeud augmente de un lorsqu'une synchronisation est effectuée. Ce mécanisme laisse donc une chance aux noeuds non joignables de revenir dans le processus mais rend prioritaires les noeuds 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 :=  $\infty$ ; 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  $\cup$  {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  $\cup$  {node, oldScore + 2 * (nbFailure + 1)}  $\setminus$  {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:    $\forall$  entry, entry ∈ currentVectorClock & entry.id == localNode.name  $\Rightarrow$  entry.v  $\leftarrow$  entry.v + 1
3:   changed  $\leftarrow$  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 nœuds, 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 événements 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 noeuds du graphe liés au groupe considéré, et soit  $V$  le nombre moyen de voisins. Chaque noeud 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 noeuds. 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 noeuds voisins pour les échanges *gossip*. La topologie est alors répartie en *slice* (sous-partie) sur différents noeuds.

La taille de la topologie due au nombre de noeuds 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 noeuds très sporadiques qui se met perpétuellement à jour. En effet la mobilité et les connexions/déconnexions rapides des noeuds 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 noeuds. 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 noeud 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 noeuds en réflexion vis-à-vis des autres noeuds du cluster. Par exemple si des noeuds A et B ne sont pas adaptables par des noeuds 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 noeuds 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 noeuds sous forme de hiérarchie dans le modèle *via* l'utilisation de plusieurs groupes. Sémantiquement les noeuds ne font pas alors partie d'un même groupe de synchronisation et exploitent alors des noeuds 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 noeuds 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 noeuds membres d'un groupe ;
- capacité à résister à la perte de connexion entre noeuds ;
- 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 noeuds logiques Kevoree déployés sur un noeud plate-forme physique de la grille de calcul. Chaque noeud logique exploite donc sa propre machine virtuelle Java sur l'implémentation du noeud JavaSE de Kevoree . La grille expérimentale exploitée pour ces expériences est composée de noeuds 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 noeuds, 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 noeud A et B, pour l'obliger à passer par un noeud 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<sup>4</sup>). 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 retracant les états temporisés des noeuds 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 événements. 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 multisaute 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.

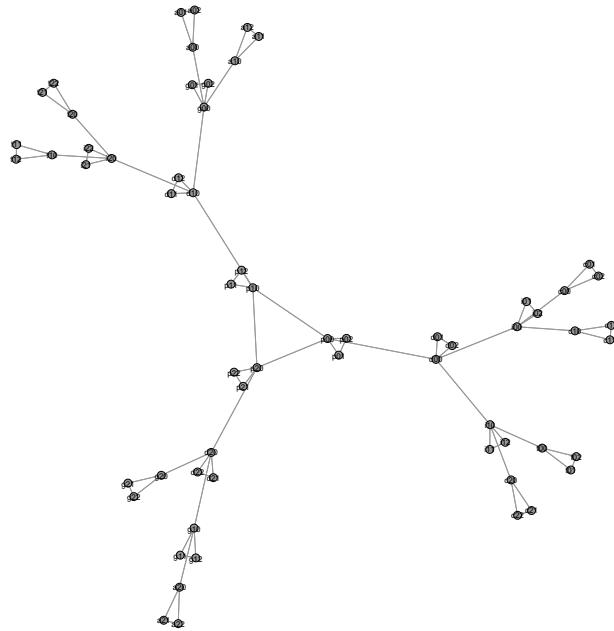


FIGURE 5.1 – Modèle de topologie, expérience #1

### 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 noeuds 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 noeud 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

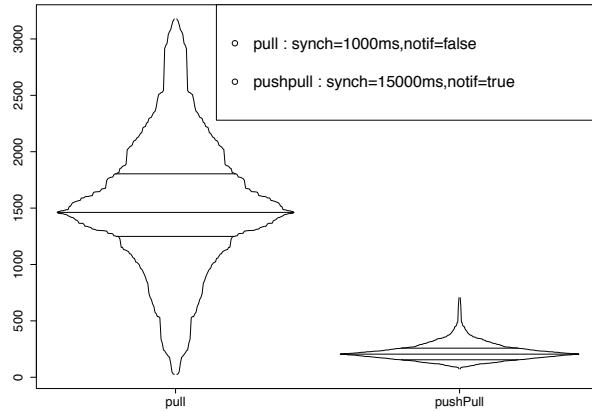


FIGURE 5.2 – Delai/hop(ms)

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.

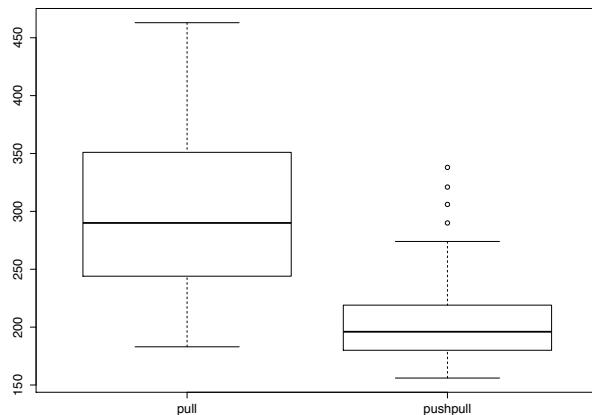


FIGURE 5.3 – Utilisation réseau/nœud(en kbytes)

## 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 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).

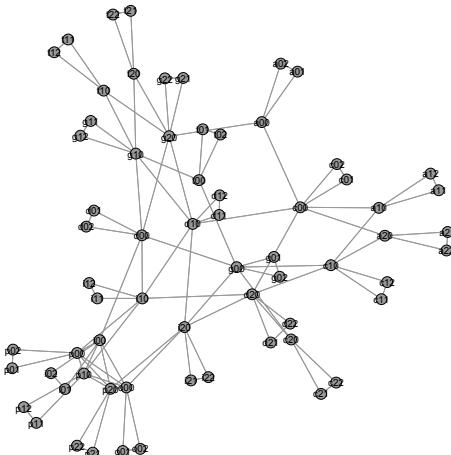


FIGURE 5.4 – Topologie pour expérience #2

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 événements 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 noeuds de communication. Cependant le mode de propagation de type *gossip* employé ici n'exploite pas ni la valeur des débits, ni les choix de noeuds 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.

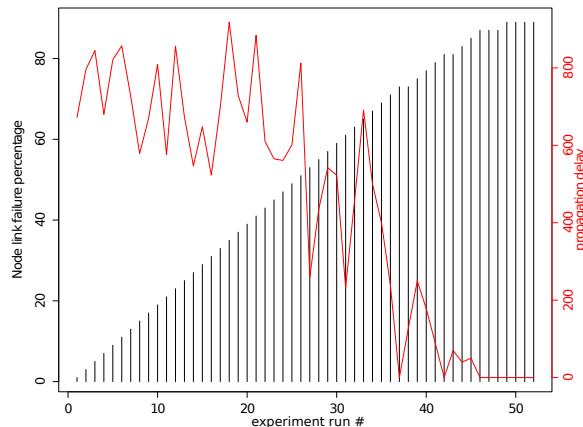


FIGURE 5.5 – Résultats expérience #2

## 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 noeuds. 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 :

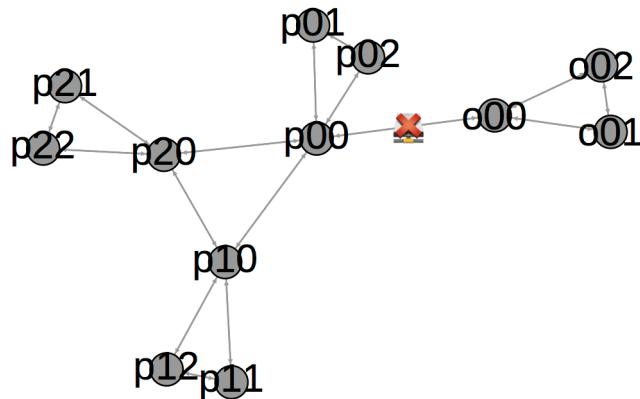


FIGURE 5.6 – Topologie pour l'expérience #3

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.

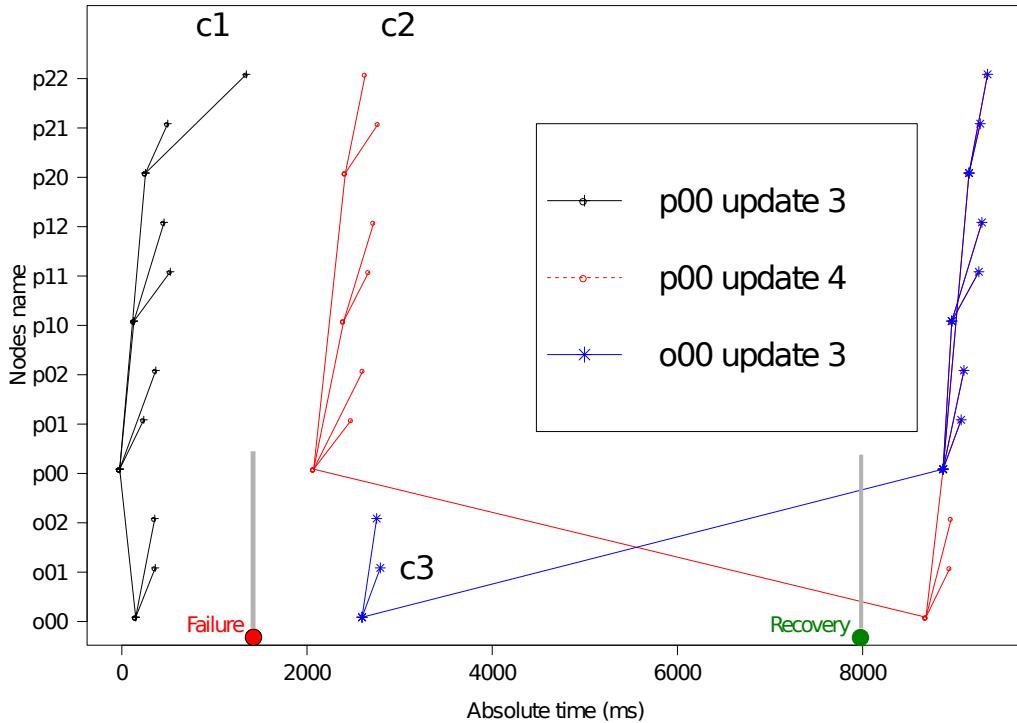


FIGURE 5.7 – Réconciliation de modèle émis en concurrence

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 autonome [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 noeuds d'infrastructure ainsi qu'un gestionnaire d'infrastructure dans le but de montrer la facilité de concevoir des nouveaux types de noeuds 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 noeud 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 noeuds pour le niveau infrastructure ainsi qu'un gestionnaire d'infrastructure capable de distribuer les noeuds de plate-forme sur les noeuds 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 noeud (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 noeuds 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<sup>1</sup>

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<sup>2</sup> de FreeBSD<sup>3</sup>. Les Jails ont été intégrées comme solution avancée de *chroot*<sup>4</sup>. 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),
})
@NoArgsConstructor
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 noeuds 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 nœuds respectivement 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<sup>5</sup>.

#### 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 noeuds dans lesquels s’exécuteraient des applications Windows. C’est pourquoi nous avons choisi de réutiliser une solution existante permettant de créer des noeuds 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 noeud possède en plus

<sup>5</sup>. <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)
})
@NoArgsConstructor
public class EC2Node extends ProxyIaaSNode {
```

---

Nous avons aussi étendu le gestionnaire de l'infrastructure pour qu'il n'envoie les noeuds de plateforme sur le Cloud EC2 que si ces noeuds 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 iaasNodesWithVirtualizationTypeNodes = getModelState.getLastModel.getNodes.filter(n =>
      KloudModelHelper.isASubType(n.getTypeDefinition, "ProxyIaaSNode"))
    if (iaasNodesWithVirtualizationTypeNodes.size > 0) {
      val potentialParents = List[String]()
      iaasNodesWithVirtualizationTypeNodes.foreach{n => potentialParents = potentialParents ++
        List[String](n.getName)}
      val parentNode = selectParent(potentialParents)
      kengine.addVariable("nodeName", node.getName)
      kengine.addVariable("parentnodeName", parentNode)
      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 noeuds 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 *JavaSENNode* 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 noeuds 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<sup>6</sup>). 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 noeud 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.

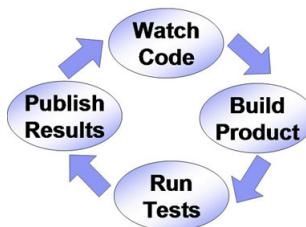


FIGURE 6.4 – Processus d'intégration continue

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<sup>+</sup>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<sup>7</sup> 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](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*<sup>8</sup> 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*<sup>9</sup> ou *TestNG*<sup>10</sup> 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'exécuteraient des esclaves 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<sup>11</sup> 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> (accessible en janvier 2013)

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<sup>12</sup> et qui est aussi proposé par le plugin maven Surefire<sup>13</sup> 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 noeuds ayant les caractéristiques associées aux tests et plusieurs tests peuvent être hébergés sur le même noeud. 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 noeud 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)
})
@NoArgsConstructor
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 noeuds disponibles et des noeuds 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 noeud de plate-forme. Pour cela, il choisit de positionner un composant et au moins l'un de ses successeurs sur le même noeud. 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 noeuds 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 successor 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<sup>14</sup>. Apache Camel est un *framework* qui implante l'ensemble des EIPs pour Enterprise Integration Patterns [HW04] en

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

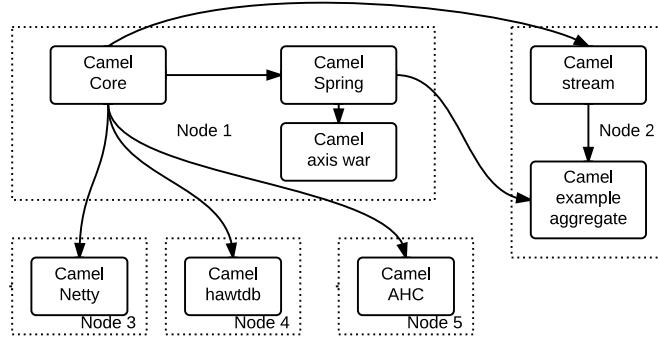


FIGURE 6.5 – Graphe de dépendances et allocation des composants sur les nœuds

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 patrons 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.

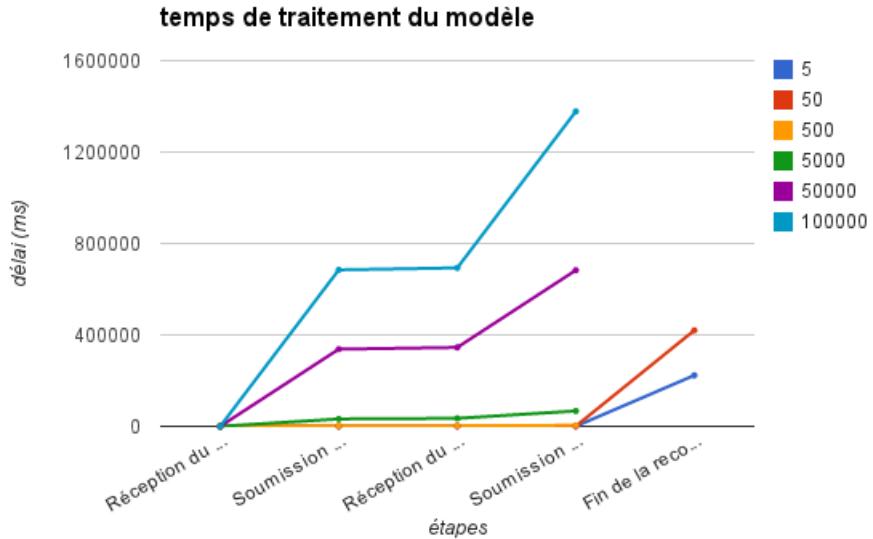


FIGURE 6.7 – Temps de traitement du modèle

Ces données nous montrent que dans le cadre du projet *Camel* (la courbe pour 50 noeuds et finissant autour des 400000 ms), le traitement du modèle utilisateur pour définir les noeuds de plate-forme utilisés pour le déploiement des tests est sensiblement le même que pour le positionnement de ces noeuds 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 noeuds 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 noeuds 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 noeuds, le temps nécessaire pour définir l'hébergement des noeuds de plate-forme sur l'infrastructure devient beaucoup plus important. Cela est principalement dû au fait de devoir, pour chaque noeud 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 noeuds 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 noeuds et 4000 éléments	3,72
500 noeuds et 40000 éléments	35,88
5000 noeuds et 100000 éléments	91,97
5000 noeuds et 200000 éléments	160,47
5000 noeuds et 300000 éléments	269,58
5000 noeuds et 400000 éléments	357,47
100000 noeuds et 400000 éléments	424,44
507000 noeuds 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 noeuds 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<sup>+14</sup>].

#### 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 noeuds de plate-forme, nous utilisons le type *PJavaSENode* ainsi que *PMiniCloudNode* qui hérite respectivement de *JavaSENode* et *MiniCloudNode*. Ces deux types de noeuds ne redéfinissent rien, que ce soit pour 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 noeuds 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 noeuds 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 noeuds 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.<sup>15</sup>

<sup>15.</sup> [cloud.diversify-project.eu](http://cloud.diversify-project.eu)

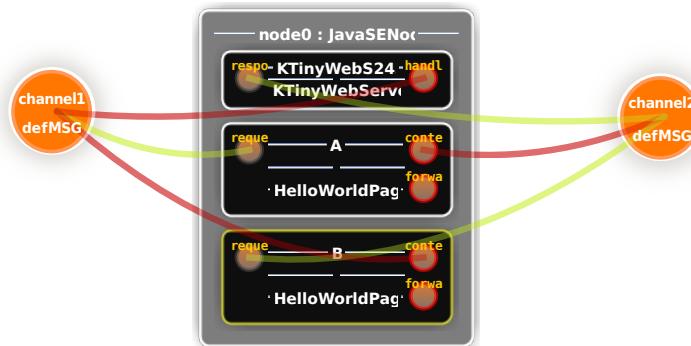


FIGURE 6.10 – Configuration de base du serveur web

**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 noeud (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
@Dictionary Type({
    @Dictionary Attribute{name = "nbNodes", defaultValue="10", optional=true},
    @Dictionary Attribute{name = "maxPercentageCPULoad", defaultValue="95", optional=true}
})
class ElasticPaaSManager extends AbstractComponentType implements ModelListener {
    // 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} : PJavaSENNode"
                    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} : JavaSENNode"
                    kengine append "addComponent {componentName}@{nodeName} : HelloHomePage"
                    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} : JavaSENNode"
                    kengine.append "addComponent {componentName}@{nodeName} : HelloHomePage"
                    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)
                            }
                    }
                }
            }
        }
    }
}

```

---

différents 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 noeud 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	Résultats et analyse expérimentale . . . . .	110
7.5.4	Extension expérimentale pour connaître 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 autonome [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<sup>1</sup> 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<sup>2</sup> 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<sup>3</sup> 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 micro-contrôleur. 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 noeuds 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 noeud Arduino peut contenir.
- **Utilisation de la mémoire persistante** : taux d'utilisation de mémoire persistante pour stocker les états du noeud. 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<sup>+</sup>12]. De manière schématique ces modèles représentent un noeud micro-contrôleur qui pilote 5 capteurs physiques et qui peut communiquer avec un noeud 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.

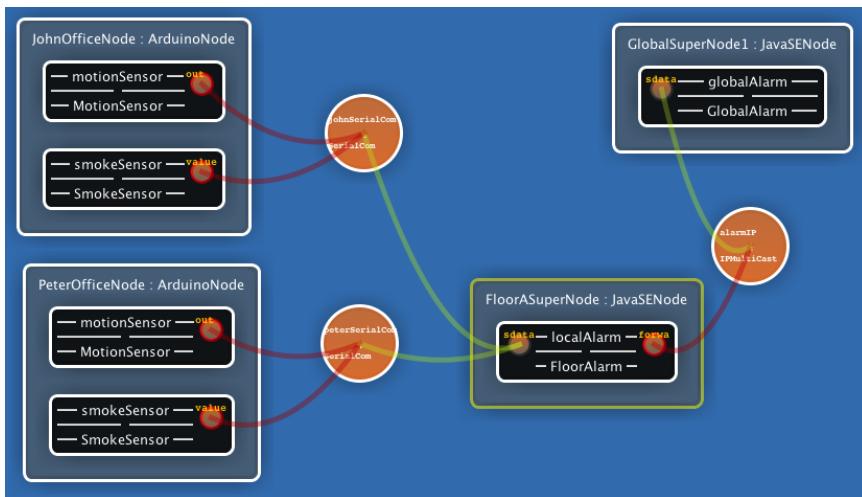


FIGURE 7.1 – Illustration modèle Kevoree de SmartBuidling

## 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

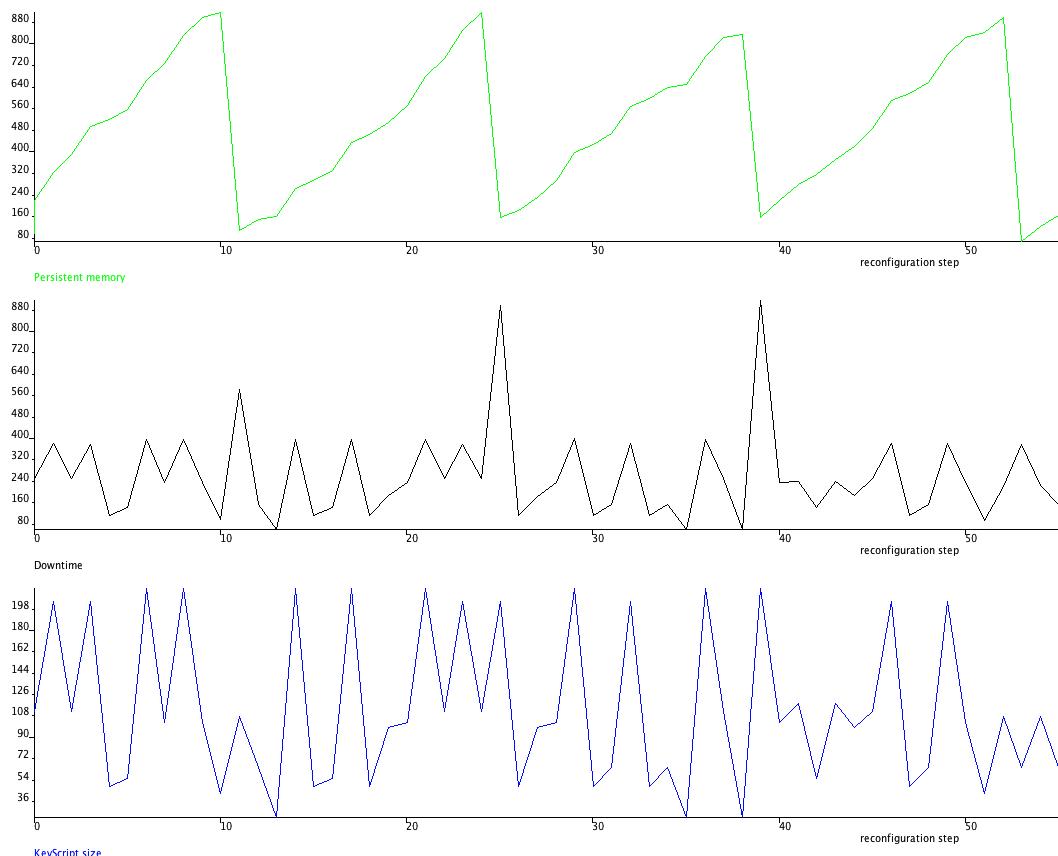


FIGURE 7.2 – Résultats expérimentaux bruts

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 Résultats 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<sup>4</sup> 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'EEPROM a 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](http://fr.wikipedia.org/wiki/Corrélation_de_Spearman)

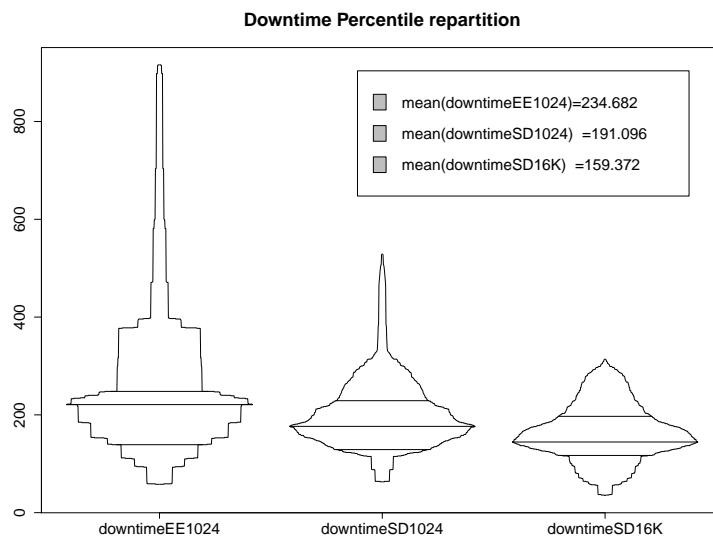


FIGURE 7.3 – Distribution percentile du temps de downtime(en ms)

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 connaître 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 discréétisation 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 additionnelles). 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

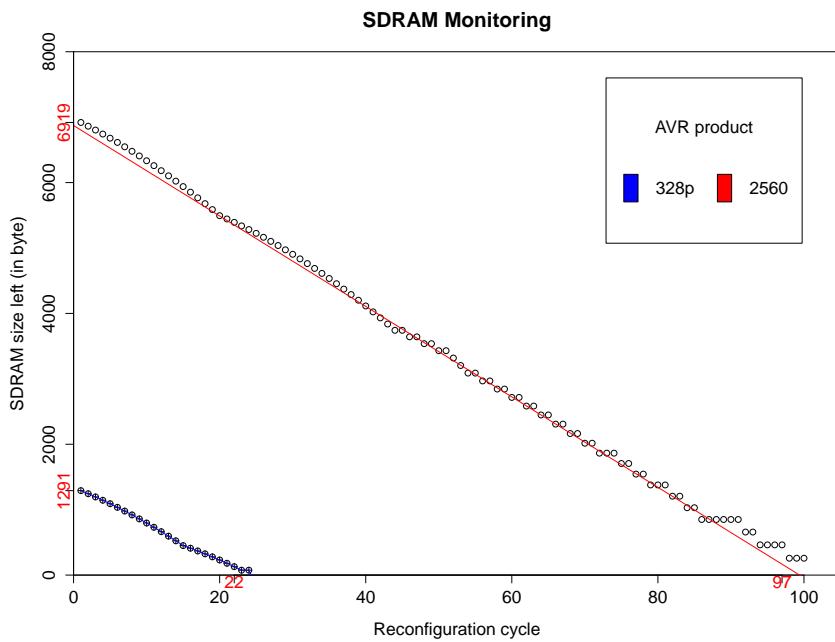


FIGURE 7.4 – Résultat expérimental sur la capacité mémoire volatile

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<sup>+</sup>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 prématuée 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<sup>5</sup>. 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.

---

<sup>5</sup>. <http://arduino.cc/en/Reference/EEPROMWrite>

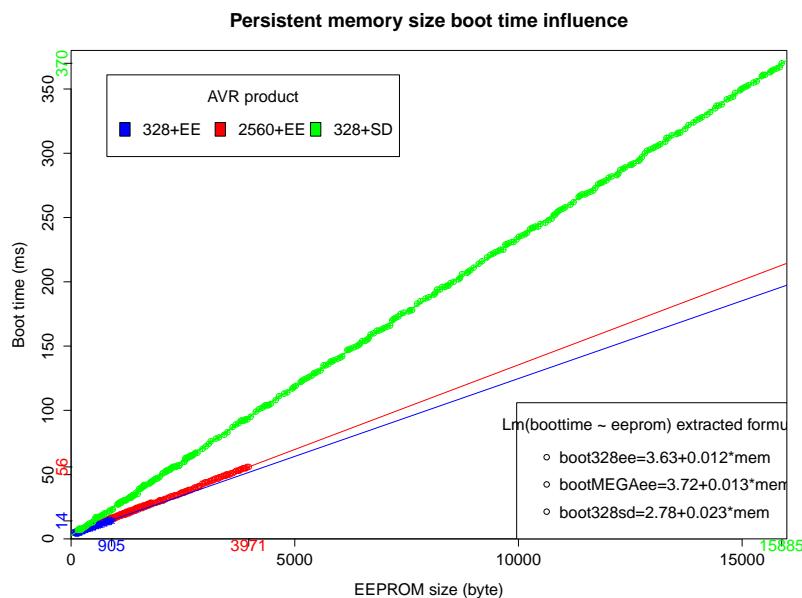


FIGURE 7.5 – Influence de la mémoire persistante sur le temps de démarrage

### 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 meilleures 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 ralenti 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 évidemment 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	<b>127</b>
8.6.1	Protocole expérimental . . . . .	127
8.6.2	Résultats expérimentaux et analyse . . . . .	128
8.7	Discussions . . . . .	<b>129</b>
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’isolation 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 à 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 spécient 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 spécient 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<sup>1</sup>. 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 nouveau type de noeud qui utilise le modèle de configuration pour guider la recherche de composants défectueux. Dans Kevoree, chaque noeud 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 noeud 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 noeud et trois nouveaux types de composants :

- le type de noeud « Nœud Contrôlé ». Ce type de noeud 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 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<sup>+</sup>14].

---

```

ranker() : list <Component>
visited = []
ranking = []
for each model M in History
    N = {c | c was added in M}
    Neighbors =  $\bigcup_{c \in N} c.neighbors$ 
    ranking.add N \ visited
    visited = visited  $\cup$  N
    SortedNeighbors = sort (Neighbors \ visited)
    ranking.add SortedNeighbors
    visited = visited  $\cup$  Neighbors
return ranking

sort (S : Set<Component>) : list<Component>
r = []
if S  $\neq$  []
    choose b | b  $\in$  S  $\wedge$  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 meilleures 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<sup>2</sup>, 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>

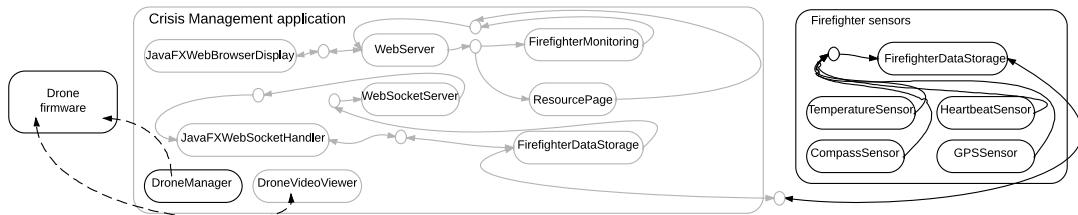


FIGURE 8.1 – Modèle de configuration pour le cas d'étude sur la gestion de crises.

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<sup>3</sup>. 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 vive 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 environnement 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<sup>+</sup>06b]. Nous avons développé un composant Kevoree pour exécuter ce benchmark<sup>4</sup>. Le type de nœud Kevoree introduit par Scapegoat a été configuré pour utiliser le mode de surveillance pleine et les contrats représentent les limites supérieures des valeurs de consommations observées<sup>5</sup>.

#### 8.4.2 Résultats expérimentaux et analyse

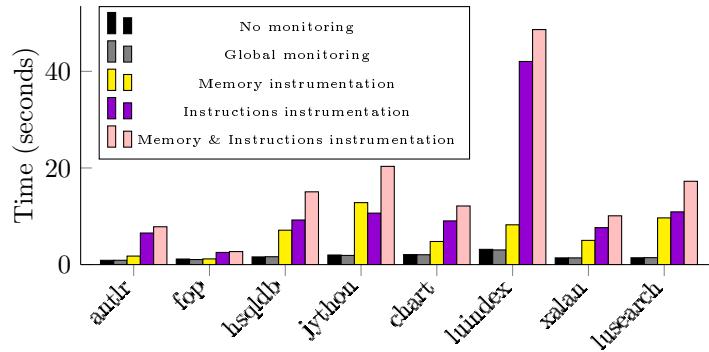


FIGURE 8.2 – Temps d'exécution observé pour les tests utilisant le benchmark DaCapo

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 :

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

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 disponibles 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 adaptive*. 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 adaptive 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 adaptive avec activation systématique de toute les sondes pour tous les composants*, *surveillance adaptive 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 adaptive 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 adaptive 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

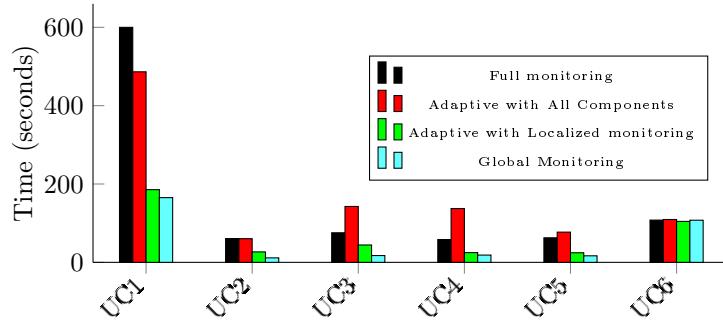


FIGURE 8.3 – Temps d'exécution pour différents scénarios d'exécution et différentes politiques de surveillance.

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 scénario 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.

$$Gain = 100 - \frac{Our\ Approach - Global\ Monitoring}{Full\ Monitoring - Global\ Monitoring} * 100$$

## 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.

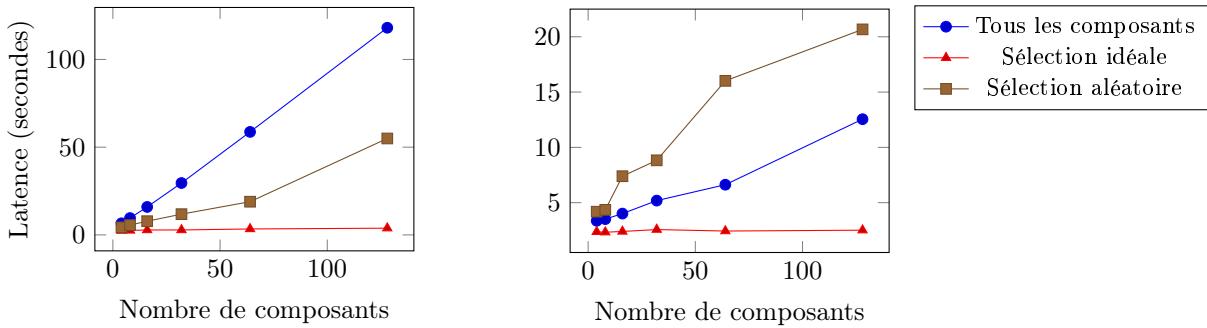


FIGURE 8.4 – Latence pour détecter un composant défectueux pour un composant construit à partir de 115 classes.

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.

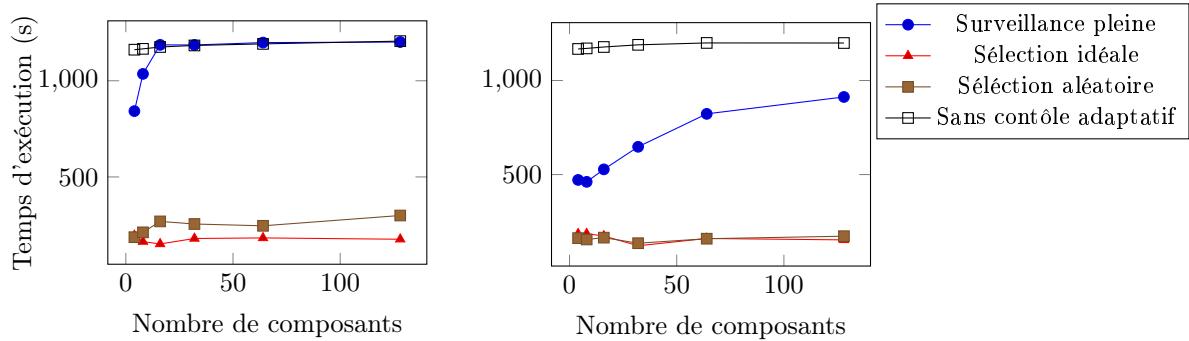


FIGURE 8.6 – Temps d'exécution pour le scénarios muni de composant composé de 115 classes.  
 FIGURE 8.7 – Temps d'exécution pour le scénarios muni de composant composé de 4 classes.

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 domaines 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 métainfirmation, 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 instantiation 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<sup>1</sup> ou Diversify<sup>2</sup>.

### 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<sup>3</sup>. 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 structurales 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.

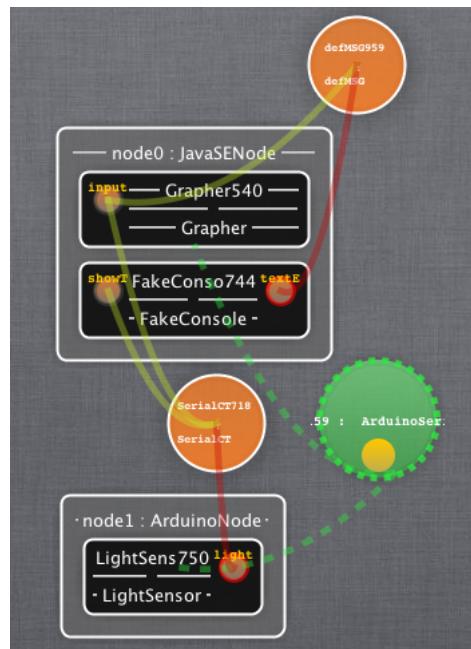
**Les Groups** sont illustrés par un rond vert et la relation avec les nœuds (abonnements) est représentée par un trait en pointillés.

**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 noeud 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 développement : 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 définition de modèle, 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<sup>4</sup> ou Kotlin<sup>5</sup> 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'outil 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 détaillera ce *framework*, qui permet de décrire les définitions de type de Kevoree sur du code source Java.

Le but de l'outil 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

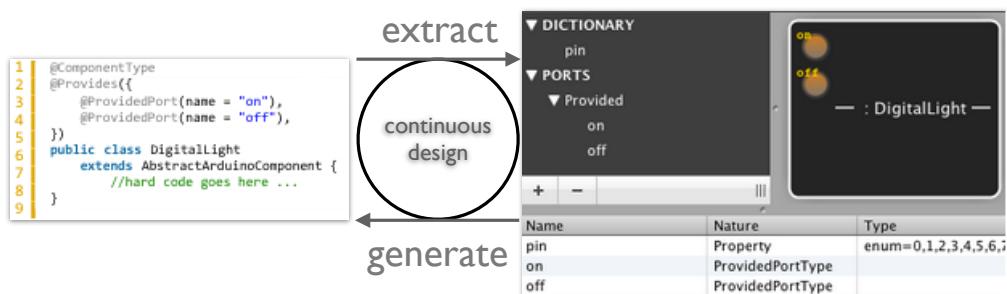
---

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 {}
@ChannelTypeFragmant
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 *DictionnaireType* 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 noeud 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 dictionnaire 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 évidemment 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<sup>+</sup>10], le modèle Kevoree doit intégrer les modèles à services et les communications par événements. 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","toggle")
@Component Type
public class FooComponentType extends AbstractComponentType {
    @Port{name="in1"}
    public void reactOnMessage(Object o){}
    @Port{name="toggle",method="toggle"}
    public boolean toggleImpl(){}
}

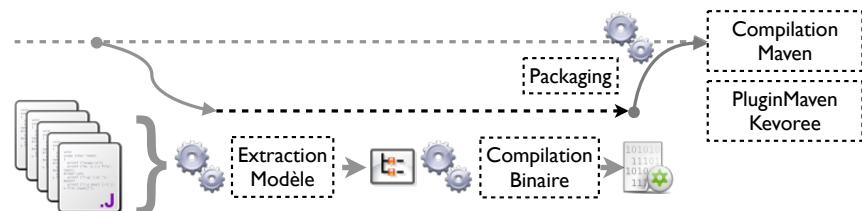
```

Listing 9.5 – Annotation de dictionnaire 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<sup>6</sup>. Le projet Apache Maven<sup>7</sup> 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/>

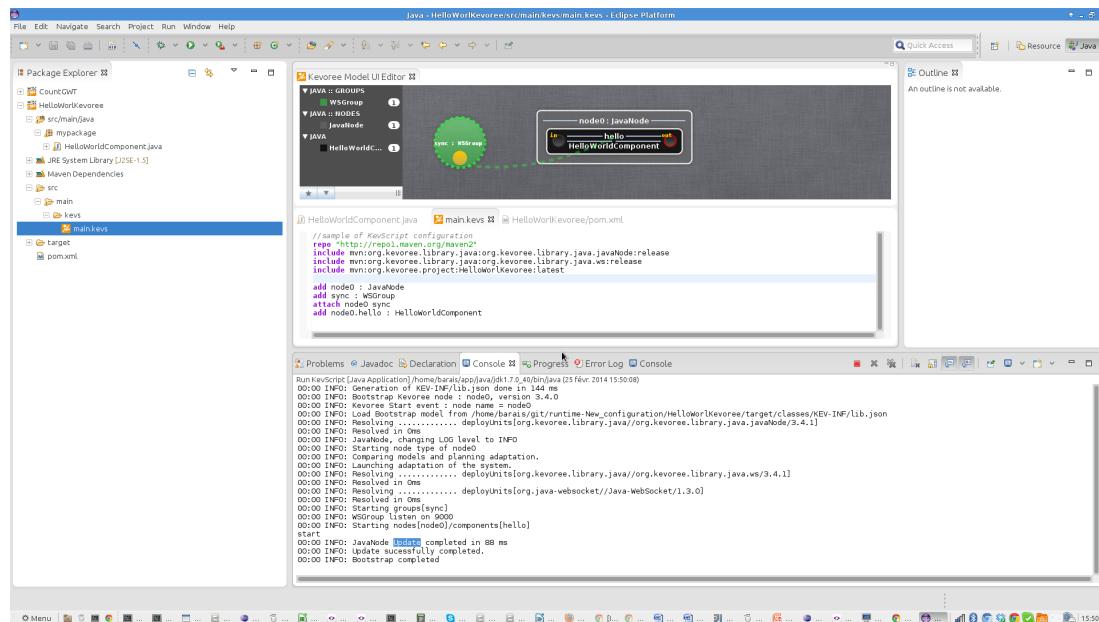
7. <http://maven.apache.org/>

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 noeuds de différentes implantations.

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



Cet environnement 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<sup>8</sup> 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*<sup>9</sup> 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-formes 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 vu dans la section 3.4 que pour chaque type de noeud, 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 noeuds 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 noeuds

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 noeuds 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 noeud 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 noeud type Kevoree OSGi ainsi développé permet de prendre comme unité de déploiement les *bundles* OSGi construits par les développeurs auxquels s'ajoutent les métainformations.

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 noeud 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 noeud 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 noeud Arduino est lui développé en C et en langage Processing<sup>10</sup>. 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 noeud parent capable de faire tourner l'algorithme de comparaison Java. Le noeud 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 noeud C++ fournit un environnement pour le développement de composants de channels, ou de groupes en C++. Construits à l'aide de la librairie boost<sup>11</sup>, 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<sup>12</sup>.

**NodeType Javascript** Le noeud 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 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<sup>13</sup>. L'implémentation est disponible ici<sup>14</sup>. Deux plate-formes distinctes sont fournies pour les composants construits pour tourner dans le navigateur et ceux les composants construits pour tourner dans un serveur d'applications comme nodeJs<sup>15</sup>.

**NodeTypes cloud** Enfin, les derniers types de noeuds 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<sup>16</sup>.

### 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 François 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<sup>17</sup>. 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 noeuds 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 noeuds 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<sup>1</sup> et du langage CVL, une initiative de standard OMG avortée. L'ensemble des contributions sur la variabilité est maintenant intégré au projet Familiar<sup>2</sup>. 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<sup>+14</sup>, VLODBH<sup>+14</sup>, FFA<sup>+13</sup>, FBA<sup>+13</sup>, FBB<sup>+12</sup>, FBLNJ12, BBLN<sup>+12</sup>, RCB<sup>+13b</sup>, RCB<sup>+12</sup>, RCB<sup>+13a</sup>, ACC<sup>+13</sup>, MBJ08, LMV<sup>+07</sup>, MVL<sup>+08</sup>, MPL<sup>+09</sup>, GVM<sup>+12</sup>].

2. Un travail sur la notion de composition de modèles avec différentes propositions dans le cadre de la modélisation par aspects [ACC<sup>+13</sup>, GVM<sup>+12</sup>, MPL<sup>+09</sup>, MVL<sup>+08</sup>, MBJ08, CBAK12, MKBJ08, MBJR07, MBB07, KKS<sup>+13</sup>, BLM<sup>+07</sup>, CBBJ08, BKB<sup>+08</sup>, Bar07, CBP<sup>+08</sup>] 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<sup>+09</sup>].

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<sup>+13</sup>, SMM<sup>+12</sup>, 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<sup>+09</sup>]. 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<sup>+12b</sup>, FDP<sup>+12a</sup>]. 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<sup>+10</sup>, NDBJ08, NBFJ09, FMF<sup>+12</sup>]. 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<sup>+</sup>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<sup>3</sup> 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 implantations de *frameworks* de modélisation [FNM<sup>+</sup>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<sup>+</sup>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
<b>Revue</b>	7	4	11
<b>Chapitre d'ouvrage</b>	3	0	3
<b>Conférence</b>	44	6	50
<b>Atelier</b>	24	6	30
<b>Rapport de recherche</b>	2	4	6
<b>Total</b>	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<sup>+</sup>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*<sup>1</sup>. Si l'on considère les 500 000 sites les plus visités<sup>2</sup>, 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'éologie 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'informatiche 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<sup>3</sup>, 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-constrainer 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<sup>+</sup>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<sup>+</sup>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*<sup>4</sup>, 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*<sup>5</sup> 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 reconfigurable 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<sup>6</sup>. 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<sup>7</sup> 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<sup>8</sup> 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<sup>9</sup>, panamax<sup>10</sup>, projectatomic<sup>11</sup>, gaudi<sup>12</sup>, mesosphere<sup>13</sup>, decking<sup>14</sup>, kubernetes<sup>15</sup> 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<sup>16</sup> [HFN<sup>+</sup>14, FNM<sup>+</sup>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 métamodélisation 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<sup>+</sup>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 reconfigurable 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 développement 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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 métamodélisation, 2007. <http://www2.lifl.fr/~mullera/CoMo07.htm> <http://www2.lifl.fr/~mullera/CoMo07.htm>.
- [BBB<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 Programming, 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 bytocode 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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 VARYability for You Workshop : Variability Modeling Made Useful for Everyone*, VARY '12, pages 3–8, New York, NY, USA, 2012. ACM.
- [FDP<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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. Marendola, N. Medvidovic, and M. Larsson, editors, *CBSE*, pages 135–144. ACM, 2012.
- [FNM<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>95] B. Hayes-Roth, K. Pfleger, P. Lalande, 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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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 on 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<sup>+</sup>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. Mikhaljlov 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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>10] G. Nain, F. Fouquet, B. Morin, O. Barais, and J.-M. Jézéquel. Integrating IoT and IoS with a Component-Based approach. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2010)*, Lille, France, France, 2010.
- [OGT<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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. Sidiropoulos-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<sup>+</sup>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<sup>+</sup>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<sup>+</sup>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.
- [VLDBH<sup>+</sup>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	Résultats 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

















































200



































































































250









































































## Cinquième partie

### CV détaillé





## 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 importants 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.