HAL
open science

Du génie logiciel pour déployer, gérer et reconfigurer les logiciels

Fabien Dagnat

To cite this version:

Fabien Dagnat. Du génie logiciel pour déployer, gérer et reconfigurer les logiciels. Interface homme-machine [cs.HC]. Télécom Bretagne; Institut Mines-Télécom, 2016. tel-01323059

HAL Id: tel-01323059

https://hal.science/tel-01323059v1

Submitted on 30 May 2016

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

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

Habilitation à Diriger les Recherches de l'Université de Rennes 1

Du génie logiciel pour déployer, gérer et reconfigurer les logiciels

Fabien Dagnat

IRISA, Équipe PASS

Soutenue le 15 janvier 2016 à Télécom Bretagne, site de Brest
devant le jury composé de

Olivier Barais	Professeur de l'Université de Rennes 1	Président
Laurence Duchien	Professeur de l'Université de Lille 1	Rapporteur
Vincent Englebert	Professeur de l'Université de Namur	Rapporteur
Daniel Le Berre	Professeur de l'Université d'Artois	Rapporteur
Frédéric Boniol	Maître de Recherche à l'ONERA et Professeur de l'INP de Toulouse	Membre
Antoine Beugnard	Professeur à Télécom Bretagne	Membre

Remerciements

Tous mes remerciements aux nombreuses personnes que j'ai croisée et avec qui j'ai travaillé. Elles ont toutes contribué à ce que je suis et donc à ce document.

Je n'oublie pas bien sûr ma famille et tous mes amis qui ont également contribué sans doute un peu moins directement...

Contenu du document

Ce document synthétise la plupart de mes activités de recherche menées depuis 2001, année de soutenance de ma thèse. Ces activités de recherche ont été conduites principalement à Brest dans le cadre des équipes Cama puis PASS du département informatique de Télécom Bretagne. L'équipe PASS est membre de l'IRISA.

Depuis septembre 2002, je suis maître de conférences à Télécom Bretagne. Ces travaux ont donc été intimement reliés et parfois ont pâti de mon autre métier à savoir celui d'enseignant. Cela m'a permis néanmoins d'être au contact de brillants étudiants qui m'ont poussé à mieux comprendre les *choses*. La notion de composant sujet d'un chapitre dans ce document en est un exemple. D'autre part, des étudiants ont participé à une partie de ces travaux en m'assistant dans le développement de démonstrateurs et la réalisation d'expériences. Un *curriculum vitæ* est inclus en annexe de ce document page 149.

Mes travaux de recherche ont également été possible grâce à divers financements :

- le projet ANR Formose, 2014-2019 ;
- le projet ANR SPaCIFY, 2006-2010 ;
- le Contrat de Recherche Externe France Télécom sauna, 2005-2008 ;
- les projets GET CARISM I en 2003 et II en 2004, CASAC en 2008 et JEMTU 2005-2008 ;
- le projet CARNOT dpan, 2010-2012 ;
- le projet région IMAJD de 2012 à 2015 ;
- le projet du conseil général LINA en 2005.

Il convient également d'ajouter la contribution de Télécom Bretagne qui m'a permis de faire un séjour d'étude chez EADS Astrium et Thalès Alenia Space d'un total de sept mois en 2009.

Durant ces années, j'ai encadré les doctorants suivants :

1. Meriem Belguidoum (11/2004 – 2/2008), *Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels*, Directeur de thèse : Guy Bernard.
2. Tuan Anh Trinh (1/2008 – 08/2011), *Déploiement distribué d'applications distribuées*, Directeur de thèse : Antoine Beugnard, thèse non soutenue.
3. Fahad Golra (1/2010 – 1/2014), *A refinement based methodology for software process modeling*, Directeur de thèse : Antoine Beugnard.
4. Xu Zhang (10/2010 – 02/2012), *A Distributed Package Administration Network*, Directeur de thèse : Antoine Beugnard, thèse abandonnée.
5. Sebastien Martinez (10/2012 –), *Impact de la mise à jour dynamique sur la chaîne de production des logiciels, et vice versa*, Directeur de thèse : Fabien Dagnat, thèse en cours.

J'ai également encadré les post-doctorants et ingénieurs suivants :

- Jérémy Buisson, (9/2007-2/2009), post-doctorant
- Eveline Kaboré, (9/2008-2/2009), post-doctorante
- Florent Diller, (10/2011-3/2012), ingénieur
- Sylvain Guérin, (2/2015-7/2015), ingénieur

D'autre part, j'ai encadré 17 stages de master recherche ainsi que de nombreux autres stagiaires :

1. David Chemouil, ENSEEIHT (2000), Docteur
2. Kamal Gakhar, Université de Paris XI (2003), Docteur
3. Meriem Belguidoum, Université de Versailles Saint Quentin en Yvelines (2004), Docteur
4. Do Manh Ha, IFI Hanoï (2004)
5. Roméo Said, Télécom Bretagne (2006), Docteur
6. Mohamed Kawtharany, Université de Bretagne Occidentale Brest (2006), Docteur
7. Benoît Vleminckx, Université de Namur (2007), Ingénieur
8. Nguyen Van Hien, IFI Hanoï (2007), Docteur
9. Tuan Anh Trinh, IFI Hanoï (2007), Ingénieur
10. Sebastien Canart, Université de Namur (2008), Ingénieur
11. Cécilia Carro, Université de Buenos Aires (2008)
12. Xu Zhang, Télécom Bretagne (2010), Ingénieur
13. Chafik Merkak, Télécom Bretagne (2011), Ingénieur
14. Chen Xi, Télécom Bretagne (2011)
15. Hanbing Li, Télécom Bretagne (2012), Thèse
16. Mehdi Alaoui Belghiti, Télécom Bretagne (2013)
17. Rachid Ayoubi, École Nationale d'Ingénieurs de Brest (2014)

Les différents travaux ont été effectués en collaboration avec de nombreux étudiants ou collègues que je tiens à remercier :

1. Anthony Abou Naoum (étudiant TB)
2. Rachid Ayoubi (master)
3. Mehdi Alaoui Belghiti (master)
4. Fan Bai (étudiant TB)
5. Meriem Belguidoum (doctorante puis collègue université de Constantine)
6. Matthieu Belin (étudiant TB)
7. Loïc Besnard (collègue IRISA)
8. Antoine Beugnard (collègue TB, éternel partenaire de discussion)
9. Jean-Paul Bodeveix (collègue IRIT)
10. Sylvain Bouveret (collègue ONERA)
11. Julien Brunel (collègue ONERA)
12. Jérémy Buisson (post-doctorant puis collègue Saint-Cyr)
13. Sebastien Canart (master)

14. Cecilia Carro (étudiante)
15. Everton Calvacante (post-doctorant)
16. David Chemouil (master puis collègue ONERA)
17. Xi Chen (master)
18. Yijun Chen (étudiant TB)
19. Jérémy Cloarec (étudiant TB)
20. Denis Conan (collègue TSP)
21. Alexandre Cortier (collègue CNES)
22. Sophie Chabridon (collègue TSP)
23. Florian Dematraz (étudiant TB)
24. Vincent Desprez (étudiant TB)
25. Florent Diller (ingénieur)
26. Florian Eizaguirre (étudiant TB)
27. Abbas Farhat (étudiant TB)
28. Ali Fendri (étudiant TB)
29. Mamoun Filali (collègue IRIT)
30. Kamal Gakhar (master)
31. Sainyam Galhotra (étudiant)
32. Johan Gall (étudiant TB)
33. Gérald Garcia (collègue Thalès Alenia Space)
34. Thierry Gautier (collègue IRISA)
35. Adrien Gentil (étudiant TB)
36. Dorian Gilly (étudiant TB)
37. Fahad Golra (doctorant)
38. Serge Guelton (étudiant TB puis collègue Quarkslab)
39. Sylvain Guérin (collègue Openflexo)
40. Christophe Guychard (collègue Openflexo)
41. Do Manh Ha (master)
42. Li Hanbing (master)
43. Jean-Baptiste Heidet (étudiant TB)
44. Hugo Hervoche (étudiant TB)
45. Lijun Huang (étudiant TB)
46. Piotr Jankowski (étudiant TB)
47. Wei Jiang (étudiant TB)
48. Evelyne Kabore (post-doctorante)
49. Mohamed Kawtharany (master)
50. Elena Leroux (collègue IRISA)

51. Étienne Louboutin (étudiant TB)
52. Marco Vereda Manchego (étudiant TB)
53. Sebastien Martinez (doctorant)
54. Chafik Merkak (master)
55. Xavier Olive (collègue Thalès Alenia Space)
56. Julien Ouy (collègue IRISA)
57. Marc Pantel (encadrant de thèse puis collègue IRIT)
58. Yue Peng (étudiant TB)
59. Luc Planche (collègue Astrium)
60. Arnaud Rinquin (étudiant TB)
61. Ana-Elena Rugina (collègue Astrium)
62. Roméo Said (master)
63. Mickaël Salaun (collègue FT R&D)
64. Arnaud Saric (étudiant TB)
65. Peiqi Shi (étudiant TB)
66. Gwendal Simon (collègue TB)
67. Martin Strecker (collègue IRIT)
68. Nicolas Szlifierski (étudiant TB)
69. Chantal Taconet (collègue TSP)
70. Jean-Pierre Talpin (collègue IRISA)
71. Tuan Anh Trinh (doctorant)
72. Nguyen Van Hien (master)
73. Benoît Vleminckx (master)
74. Xu Zhang (doctorant)

Table des matières

1 Introduction **1**
1.1 Le contexte ... 1
1.2 Démarche ... 2
1.3 Plan du document ... 3

2 La dépendance pour gérer les logiciels **5**
2.1 Motivation ... 5
2.2 Des logiciels à base de composants ... 7
2.3 Des descriptions mathématiques ... 9
2.3.1 La dépendance ... 9
2.3.2 Le contexte ... 10
2.4 Des règles mathématiques de déploiement ... 12
2.4.1 L'installation ... 12
2.4.2 Description de la désinstallation ... 15
2.4.3 Mise à jour des composants ... 17
2.5 Extension avec des propriétés ... 19
2.5.1 Propriétés et contraintes ... 19
2.5.2 Impacts sur le système ... 20
2.6 Preuve de correction ... 21
2.7 Discussion et perspectives ... 22

3 La notion de composant logiciel **25**
3.1 La notion de composant ... 25
3.2 Généralités ... 26
3.2.1 Les éléments architecturaux ... 27
3.2.2 Variabilité au cours du processus de développement ... 28
3.3 Qu'est-ce qu'un composant ? ... 32
3.4 Qu'est-ce qu'un connecteur ? ... 36
3.5 Qu'est-ce qu'une architecture ? ... 39
3.6 Discussion ... 41

4 Le déploiement **45**
4.1 Processus de déploiement ... 45
4.1.1 La motivation ... 46
4.1.2 La collecte des exigences de déploiement ... 48
4.1.3 Le déploiement ... 50
4.1.4 Un exemple ... 52

4.1.5 Le prototype ... 54
4.2 Le déploiement réparti pair à pair ... 57
4.2.1 Le déploiement actuellement ... 57
4.2.2 Vers un déploiement pair-à-pair ... 61
4.2.3 Une distribution pair-à-pair pour quels usages ? ... 62
4.2.4 Un format de méta-donnée pour `dpan` ... 63
4.2.5 Les fonctionnalités de `dpan` ... 66
4.2.6 Une simulation de `dpan` ... 68
4.3 Bilan ... 69

5 La mise à jour à chaud de logiciel 71
5.1 Les problématiques de la mise à jour à chaud ... 72
5.1.1 Généralités ... 72
5.1.2 Les problématiques de la reconfiguration ... 73
5.1.3 Quelques propositions ... 75
5.2 Une méthode pour la mise à jour à chaud de satellites ... 78
5.2.1 Contexte ... 78
5.2.2 Les pratiques à l'époque du projet ... 80
5.2.3 La proposition d'ingénierie des reconfigurations ... 81
5.3 ReCaml ... 83
5.3.1 Motivation ... 83
5.3.2 Les principes de ReCaml ... 83
5.3.3 Un exemple d'utilisation de ReCaml ... 84
5.3.4 La démarche et langage ... 86
5.4 Pymoult ... 91
5.4.1 Un cadre général pour le DSU ... 92
5.4.2 Des choix variés et figés par plate-forme ... 93
5.4.3 Vers une API générique pour le DSU ... 97
5.4.4 Des expérimentations nombreuses ... 103
5.5 Pycots et Coqcots ... 104
5.6 Bilan et perspectives ... 108

6 Des processus comme architecture d'activité 111
6.1 Motivation ... 111
6.2 Les limites des approches actuelles ... 112
6.3 Une approche multi-méta-modèle ... 114
6.4 Les méta-modèles pour le développement de processus ... 115
6.4.1 Le méta-modèle de spécification ... 116
6.4.2 Le méta-modèle d'implémentation ... 116
6.4.3 Le méta-modèle d'instanciation ... 117
6.5 Un environnement de spécification et d'exécution ... 119
6.6 L'évaluation de l'approche ... 121
6.7 Bilan et perspectives ... 123

7 Bilan et perspectives 125
7.1 Bilan ... 125
7.2 Perspectives ... 128

7.2.1 Fédération de modèles 128
7.2.2 Des outils transverses pour le déploiement 129
7.2.3 Vers un déploiement dynamique transparent 131

Bibliographie **133**
Publications personnelles 133
Publications d'autres auteurs 138

Curriculum Vitæ **149**
État civil 149
Contact 149
Emplois 149
Formation 149
Recherche 149
Projets 150
Encadrements 150
Conférences et Expertises 152
Jurys et Comités 152
Publications Majeures Récentes 152
Responsabilités 153
Enseignements 153

Chapitre 1

Introduction

1.1 Le contexte

Comme tout cours de *Génie Logiciel* aime le rappeler, il s'agit d'une discipline d'ingénierie née à la fin des années soixante-dix pour tenter de remédier à un constat généralisé d'échec de la plupart des projets informatiques. Cette discipline a été parcourue par de nombreux courants mais reste cohérente par l'objectif affiché. Ainsi, beaucoup d'efforts ont conduit à la proposition de nombreux cadres mathématiques, de nombreuses méthodes et de nombreux outils pour aider à la production, de manière efficace et peu coûteuse, de logiciels.

Dans ce cadre, le logiciel en construction est devenu un sujet d'étude précis et son développeur peut donc s'appuyer sur de multiples contributions pour améliorer ou faciliter son travail. On a ainsi vu apparaître une certaine industrialisation du processus de production de logiciel qui permet aujourd'hui de produire, chaque jour, des quantités astronomiques de logiciel. À ma connaissance, il n'existe pas de chiffres disponibles mais certaines statistiques ne trompent pas :

- Une distribution Linux comme Debian contient plusieurs dizaines de milliers de paquets logiciels [1] que certains ont mesuré en ligne de code. On trouve ainsi, 419 776 604 lignes de code pour la distribution Debian en février 2012 [2].
- On estime à environ mille nouvelles applications par jour qui seraient soumises sur l'AppStore d'Apple [3].
- Les taux de croissance de certains logiciels sont connus et plutôt importants. Le noyau Linux par exemple a grossi de 288 % entre les versions 2.2 et 2.6 puis à nouveau de 288 % entre la version 2.6 et 3.1. Pour Windows, de NT3.1 à Vista, il y a six versions pour une croissance cumulée de 1131 % [4].

Le logiciel est donc devenu un produit de grande consommation qui est produit vite et en grande quantité. Et ceci aussi parce que les machines informatiques ont envahi notre quotidien sous de nombreuses formes (ordinateur classique, portable, tablette, téléphone, télévision, réfrigérateurs, voiture, ...). En tant qu'utilisateurs de ces multiples machines, nous souhaitons tout faire avec des logiciels : des tâches usuelles comme gérer ses contacts ou son agenda aux nouveaux usages émergents du téléphone comme payer ses achats ou suivre ses progrès en course pour un athlète.

1. Pour Ubuntu, un simple `wc -l` sur la liste des paquets vaut 54 673, http://packages.ubuntu.com/utopic/allpackages?format=txt.gz.
2. https://goo.gl/YMmIbx
3. http://goo.gl/XwbiEH
4. http://www.informationisbeautiful.net/visualizations/million-lines-of-code

Ainsi chacun veut toujours plus de logiciels adaptés à ses besoins et s'exécutant sur tous les équipements connectés qu'il possède.

Ce qui est surprenant est que pendant longtemps, le monde de la gestion des logiciels après leur production est resté dominé par des pratiques manuelles de recherche dans des listes, de téléchargement unitaire et d'installation manuelle. C'est dans ce cadre que j'ai développé une activité de recherche visant à fournir des cadres mathématiques, des méthodes et des outils pour gérer massivement les logiciels. Ici, la gestion recouvre toutes les opérations que l'on peut vouloir faire avec des logiciels, une fois qu'ils ont été produits : déployer, valider, diffuser, vérifier, exécuter ou mettre à jour. Il est entendu que mes travaux portent sur la gestion du logiciel une fois développé mais que mes propositions ont des impacts sur la façon de produire un logiciel. Ainsi, par exemple, pour le déploiement, nous proposons des langages de description des dépendances qui doivent être utilisés durant la production du logiciel. Pour la reconfiguration certains mécanismes imposent également de structurer son code de la manière qui convient.

1.2 Démarche

La tradition scientifique repose sur une méthode universelle pour résoudre les problèmes :

1. Réduire le problème à un cadre précis.
2. Identifier les concepts d'importance dans ce cadre.
3. Analyser et expérimenter ces concepts afin de proposer des solutions et outils correspondant au cadre identifié en 1.
4. Mettre œuvre et utiliser la proposition.
5. Identifier les limites de la proposition et du cadre.
6. Tenter de généraliser la proposition à un cadre plus large.

Dans la pratique actuelle de la recherche, le point 6 est peu fait ou alors en restant dans le cadre d'une communauté et en faisant de petits pas.

Le problème au centre de ma recherche autour de la manipulation de logiciels est complexe. Elle nécessite de nombreuses applications de la démarche précédente dans de nombreux cadres et confronter, faire interagir les différentes propositions de façon à ce qu'elles s'enrichissent mutuellement. Cette idée a été directrice dans mon travail de recherche et ce depuis mes travaux de thèse. J'ai travaillé à la croisée de plusieurs communautés et domaines de façon à réutiliser les concepts, les croiser et essayer de les enrichir et de les généraliser. Par exemple, on pourra citer deux exemples de telles propositions, que j'estime intéressantes, de ce type de croisement :

- La notion de composant pour la définition de processus de développement. Pour introduire de la souplesse dans la construction de modèle de processus de développement et permettre une certaine adaptation durant leur « exécution », nous avons proposé de réutiliser la notion de composant, une activité devient un composant avec une spécification (son type) et son implantation. Ainsi, réutiliser dans la spécification d'activité est plus simple puisque l'on peut redéfinir l'implantation à sa guise pour l'adapter à ses besoins. On peut aussi, à l'exécution, remplacer une implantation par une autre si elles satisfont la même spécification abstraite (elles sont du même type). Le chapitre 6 page 111 décrit en détail mes travaux dans ce domaine.
- Suivre une approche fonctionnelle dans le cadre de la modélisation pour aller au-delà du cadre objet. En effet, la communauté des modèles (à la UML) est tellement centrée sur la

notion d'objet que certaines propositions intéressantes lui échappent. Ainsi, par exemple, dans des recherches sur la modélisation, on essaye d'importer des concepts qui ont prouvé leur intérêt dans le cadre de la programmation fonctionnelle. Concrètement, la paresse est une notion intéressante au cœur du succès d'Haskell et qui, j'en suis persuadé, pourrait contribuer à des améliorations des techniques et outils de modélisation. Une première expérience a été réalisée et publiée dans [12], elle mériterait d'être poursuivie. D'autre part, dans le cadre d'une reflexion menée autour du *framework* de modélisation OpenFlexo [5], nous essayons d'utiliser des langages fonctionnels réactifs pour construire des environnements de modélisation. Des premiers éléments ont été présentés aux journées Neptune et décrits dans [8].

Dans cette démarche, une partie difficile est de bien identifier les concepts par rapport à un cadre précis sans se laisser berner par un nom que l'on retrouve souvent dans de nombreux cadres mais qui désigne à chaque fois des « choses » différentes. Par exemple, le terme *composant* recouvre de nombreux concepts (un composant de conception n'a rien à voir avec un composant que l'on déploie) comme nous le verrons en détail en partie dans le chapitre 3 page 25. On pourrait également citer les termes de modèle ou langage qui eux aussi sont utilisés dans de nombreux cadres pour désigner des notions différentes. La tentation est souvent d'essayer d'unifier ou d'aligner et ainsi de fixer le sens d'un terme dans différents cadres. Comme parfois les notions sont contradictoires, l'unification ou l'alignement ne semblent pas toujours possible. Et même lorsqu'ils le sont, on peut se poser la question de l'intérêt de l'opération, ne faudrait-il pas plutôt maintenir des liens (des points communs ou des contradictions par exemple) ? Dans mes travaux de recherche, cette démarche a été centrale. Pour la modélisation, le terme de fédération de modèles est utilisé pour la décrire. Ainsi, un modèle fédéré est un modèle construit à partir de liens vers de multiples modèles pré-existants [27]. Attention, ici, un lien peut avoir une sémantique complexe. Par exemple, on peut vouloir dire « copie de la valeur de cette information le 17/10/2014 » mais aussi « toujours la même valeur » ou bien encore « à chaque fois que telle autre information change de valeur, recopier la nouvelle valeur de l'information liée ». Cette notion de fédération est encore jeune mais semble prometteuse. Elle tire une partie de mes activités vers la modélisation dans le cadre de mon interaction avec la *start-up* Openflexo. Elle a donnée lieu récemment aux publications suivantes [7, 22, 27, 8]. J'ai néanmoins choisi de ne pas couvrir cette partie de mes recherches dans ce document. En effet, il est encore en plein bouillonnement et je manque de recul sur son devenir.

1.3 Plan du document

La rédaction de ce manuscrit s'est déroulée sur une longue période. Il est donc possible que malgré mes efforts pour mettre les différents éléments en cohérence et en relation, ils paraissent un peu disparates. Je m'en excuse, par avance, auprès du lecteur.

J'ai choisi de découper ma présentation en cinq parties. Chacune va faire l'objet d'un chapitre. La logique dans ce découpage a été thématique, les chapitres seront donc les suivants :

– Le premier chapitre se centre sur mon activité dans le cadre de la gestion des dépendances de logiciels lors de leur déploiement. Ce chapitre est le plus formel des activités présentées dans ce manuscrit. Il s'agit ainsi de synthétiser les modèles mathématiques et les algorithmes associés que nous avons proposé pour garantir la sûreté du déploiement. Ces travaux ont principalement eu lieu entre les années 2004 et 2009.

5. http://openflexo.org

- Le deuxième chapitre reste théorique mais est moins formel, il aborde la question des modèles de composants logiciels. La contribution me semble plus d'ordre méthodologique et terminologique. Ces travaux sont essentiellement réalisés entre 2007 et 2011.
- Le troisième chapitre regroupe des efforts différents autour de la notion de déploiement en général. Une première partie porte sur le processus de déploiement dans un cadre distribué étudié entre 2007 et 2011. La seconde porte plutôt sur les conséquences de l'explosion de quantité de logiciel produit et sur les outils qui seront necéssaires. Ces travaux s'étendent sur la période 2009-2012.
- Le quatrième chapitre, plus long que les autres, synthétise mes travaux depuis 2007 sur la mise à jour à chaud de logiciels. Il s'agit donc de mettre à jour des logiciels pendant qu'ils s'exécutent.
- Le cinquième chapitre, le plus court, décrit mes activités de recherche dans le cadre de la conception et la réalisation de processus de développement logiciel. Ces travaux ont eu lieu sur la période 2009-2014.

Tous ces travaux se nourrissent et se complètent. Ainsi, mes travaux sur le déploiement m'ont poussé à la proposition de modèle de composant multi-méta-modèle. En séparant les différentes préoccupations, il est possible de proposer des méta-modèles puissants qui restent utilisables. La complexité des opérations de déploiement et surtout des processus de collecte des contraintes de déploiement au cours du cycle de développement du logiciel, m'ont poussé à investiguer la façon de les décrire. Combiner avec une approche multi-méta-modèle cela a conduit à mes travaux sur les processus logiciels.

Ces différents rapprochements et liens ont bien sûr été, pour beaucoup, identifiés et reconstruits lors de la rédaction de ce manuscrit. Ils sont cycliques puisque toutes ces activités étaient menées souvent de front. Le document (et donc son plan) essaye donc de les aplatir pour fournir un ordre de lecture logique.

Enfin, pour faciliter la distinction, tout au long du manuscrit les citations de mes articles sont simplement numérotées alors que les publications des autres auteurs commencent par un P avant leur numéro d'ordre. Ainsi, le lecteur pourra clairement distinguer les citations personnelles des autres.

Chapitre 2

La dépendance pour gérer les logiciels

Ce chapitre synthétise des travaux de recherche menés entre 2004 et 2009. Dans le cadre, de ces travaux, j'ai encadré la thèse de Meriem Belguidoum qui est maintenant maître de conférences à l'université de Constantine. Ces travaux ont été publiés dans diverses conférences et revues [33, 23, 4, 17, 5, 18, 24]. Les principales contributions de cette activité de recherche sont :

- Un modèle mathématique simple et complet de la notion de dépendance et des actions de déploiement.
- Un algorithme qui permet d'assurer le déploiement sûr de composants logiciels.
- Une preuve (manuelle) de la correction des algorithmes proposés.
- Une implantation en Caml de notre algorithme de vérification de déploiement. Cet outil a été utilisé dans le cadre de la réalisation d'un prototype de déploiement d'applications réparties développées en Fractal ou utilisant OSGi.
- Des premières expériences sur l'extension de notre modèle inhéremment centralisé au support d'applications réparties.

2.1 Motivation

Les pratiques de déploiement logiciel se sont consolidées durant une période pendant laquelle l'informatique était organisée de manière assez simple. À cette époque, figure 2.1, plusieurs utilisateurs (en bleu) partageaient une machine, plutôt puissante qui était administrée par des professionnels (ici en rouge). Toute opération de déploiement passait par le truchement des administrateurs qui avaient toutes les compétences (et le temps) pour réaliser l'opération en question en s'assurant de sa réussite et de la non perturbation de la machine hôte. De plus, ces machines appartenaient à un seul réseau et ce de manière fixe.

L'administration des logiciels était relativement simple et assuré par des professionnels formés et compétents. Mais, l'usage des équipements informatiques a été totalement bouleversé.

Maintenant, figure 2.2, la vie quotidienne des usagers du monde digital mêle des usages professionnels avec la vie quotidienne aussi bien en terme d'équipements (utilisation de son *smartphone* personnel, utilisation pour les loisirs de son portable de travail, ...), de réseau d'accès que d'applications (comment mélanger son agenda, ses contacts, ...). Dans ce contexte,

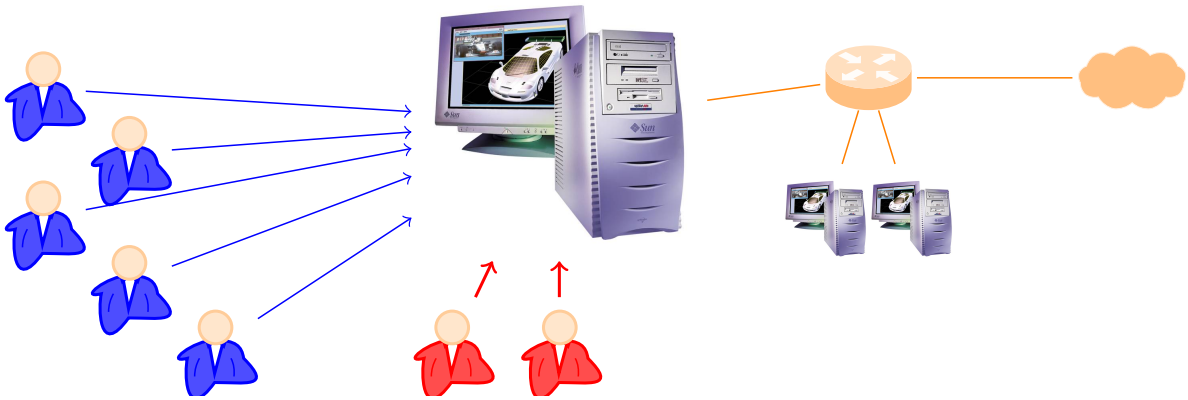


Figure 2.1 – Le déploiement de logiciel, avant

le nombre d'équipements explose, ces équipements changent sans arrêt de contexte, de réseau d'accès et sont administrés par de multiples personnes dont la majorité n'y connait pas grand chose.

La complexité grandissante du monde digital a aussi un effet indirect un peu moins visible au premier abord par les utilisateurs mais que tous les développeurs connaissent bien. Les logiciels doivent supporter des tâches de plus en plus complexes, s'intégrer au mieux au sein de l'ensemble des applications de l'utilisateurs et s'adapter à ses contextes d'usage. Un logiciel est maintenant un objet complexe reposant sur de nombreux cadriciels, réutilisant et étendant de nombreux autres logiciels et utilisant de nombreux services en ligne. Enfin, il doit s'adapter sans sourciller à tous les changements de contexte imposé par son utilisateur.

Dans ce nouveau contexte, nous pensons que la gestion de logiciel doit être simplifiée au maximum et doit être assurée de manière sûre. C'est-à-dire, l'outil de déploiement doit s'assurer que les opérations de déploiement qu'il réalise ne vont pas rendre inutilisable l'équipement en question. Il est clair qu'il faudrait ajouter une autonomie [P65] suffisante à ces systèmes pour qu'ils puissent réagir, s'adapter aux différents changements de leur environnement. Cette adaptation doit être réalisée tout au long du cycle de vie du déploiement [P26]. Ainsi, les systèmes et les réseaux informatiques sont encore installés, déployés, administrés avec des techniques qui ne sont pas à la mesure de leur complexité. À ce jour, aucune méthode et aucun outil général ne permet de gérer complètement le déploiement au sein d'une entreprise. L'intervention humaine est encore cruciale et consommatrice en temps. Ces outils ne sont encore que des aides aux tâches des administrateurs systèmes qui, certes, facilitent leur travail mais ne diminuent pas vraiment le coût de gestion du parc informatique (aussi bien matériel que logiciel) d'une entreprise. Notons également qu'aucune des solutions proposées aux professionnels ne s'accommodent des outils personnels et donc de la gestion d'équipements à usage mixte.

Hors du monde professionnel, l'utilisateur est souvent laissé un peu à l'abandon et doit se contenter des outils choisis par l'éditeur de son système d'exploitation qui laissent peu de place aux logiciels des concurrents. Dans ce cadre, avoir des équipements de plusieurs constructeurs / éditeurs peut se reveler un casse tête pour gérer ses différentes applications même lorsqu'elles peuvent s'exécuter sur plusieurs plates-formes. Enfin, même dans un environnement mono-équipementier, rien ne permet de gérer les logiciels de son parc de machine pour un particulier

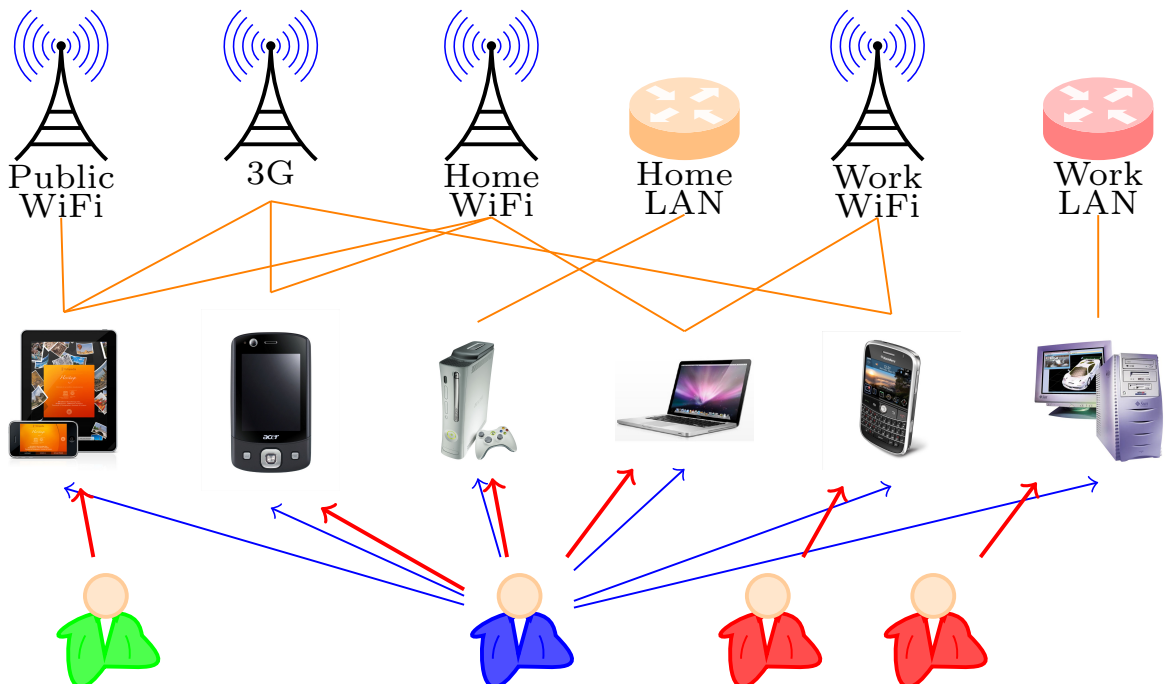


Figure 2.2 – Le déploiement de logiciel maintenant

non initié à l'informatique. Il faudra la plupart du temps passer du temps sur chacun des équipements pour installer un logiciel ou mettre à jour ses applications.

Le dernier effet qui concourt à cette complexité vient du fait que chaque application est maintenant complétée par des extensions, les fameux *plugins* qu'il faut aussi gérer. Ainsi, l'utilisateur peut avoir un nombre important de plates-formes de déploiement. À titre personnel, je suis contraint d'utiliser les plates-formes de Firefox, Eclipse, Latex (`tlmgr`), homebrew (des logiciels unix pour mac), `opam` (les outils et librairies Caml), `cabal` (Haskell), `cpan` (Perl), rubyGems, `easy_install` et `pip` (Python), maven utilisé par un certain nombre d'applications Java, les divers gestionnaires de paquets pour Emacs (elpa, marmalade et melpa), le *mac app store*, certains logiciels qui ont leur propre système... Soit plus de quinze plates-formes, avec toutes des possibilités différentes et des modes d'interaction spécifiques, qui consomment une énergie non négligeable de ma part.

Pour aboutir à un système autonome et sûr de déploiement de nombreux travaux de recherche sont encore indispensables. Le principal effort consiste à passer d'une approche pragmatique proposant des solutions ad-hoc adaptées dans certains cas, à une véritable approche industrielle, générique et flexible basée sur des concepts clairs qui permettra de maîtriser le déploiement. À partir de là, il sera alors possible d'envisager de conférer une autonomie à ces systèmes.

2.2 Des logiciels à base de composants

L'approche usuelle consiste à assembler les composants (les connecter ou déconnecter) de manière explicite et manuelle. Malheureusement, cette façon d'assembler ne peut pas vraiment passer à l'échelle et ne permet donc pas de construire un système complet (au sens de tous les logiciels d'une machine) à base de composants. Notre idée, consiste à utiliser l'approche

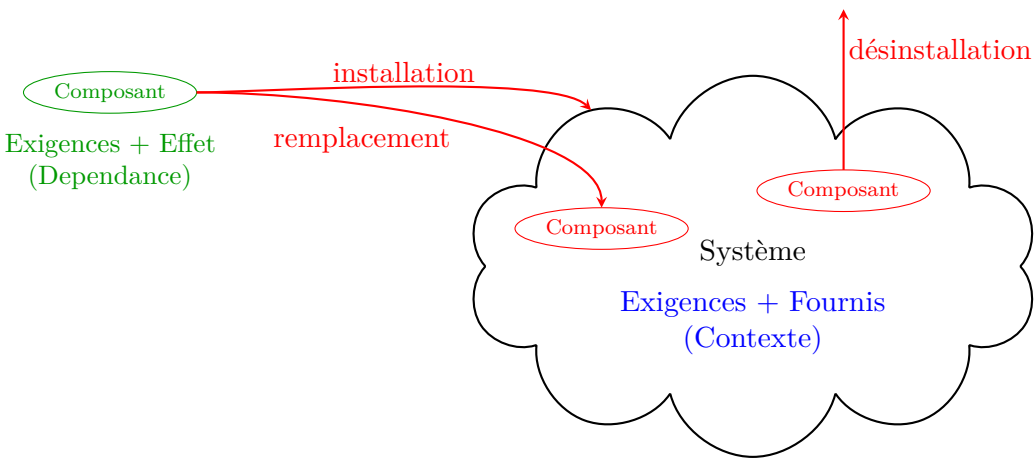


Figure 2.3 – Le déploiement de composants

suivie par les gestionnaires de paquetages logiciels dans le cadre des composants logiciels, cela permettra de bénéficier de la maturité de ces gestionnaires au niveau de l'expressivité des exigences et des besoins, en les adaptant à la structure des composants. La connexion des composants devient donc dirigée par des contraintes plutôt que d'être explicite. La difficulté de cette adaptation réside dans le fait qu'un paquetage fournit un service unique (lui-même) et ne peut donc pas avoir la structure d'un composant qui peut fournir de nombreux services.

Nos travaux ont permis de proposer un modèle mathématique simple. Celui-ci correspond au schéma de la figure 2.3. Le système, dans lequel nous déployons, abstrait une machine sous la forme d'un ensemble de composants (fournis). Ainsi, l'installation d'un nouveau composant, la désinstallation ou la mise à jour d'un composant installé est modélisée respectivement par l'ajout d'un composant dans le système, le remplacement ou le retrait d'un composant actuel du système. Pour ce faire, nous utilisons la description des exigences du composant ainsi que la description de l'effet de son installation dans le système. Une fois, que le système de déploiement peut prouver que l'opération en question est correcte, il peut déclencher les mécanismes réels de déploiement comme illustré dans la figure 2.4 page ci-contre. Nous séparons, ainsi, la réalisation concrète du déploiement de sa description abstraite et sa vérification formelle. Il est ainsi possible de réutiliser directement les mécanismes de déploiement de bas niveau fournis par les différentes plates-formes.

Pour assurer la vérification du déploiement, nous utilisons des règles de déploiement qui décrivent formellement les conditions de *bonne* réalisation. Si le système peut construire une preuve de déploiement correct, l'opération est acceptée. Il y a deux catégories de règles : celles qui vérifient la possibilité du déploiement (*i.e.* est-il possible de déployer une certaine entité dans un contexte particulier ?) et celles qui calculent l'effet du déploiement sur le système cible (contexte) une fois que l'opération de déploiement est possible.

L'installation, la désinstallation et la mise à jour d'un composant dans un système correspondent respectivement à son ajout, sa suppression et son remplacement dans le système. Le succès de ces opérations exige le respect des contraintes suivantes :

1. le système fournit les ressources exigées et les services requis par le composant à installer ;
2. le composant à installer et ses services fournis ne sont pas en conflit avec les composants et les services déjà installés ;

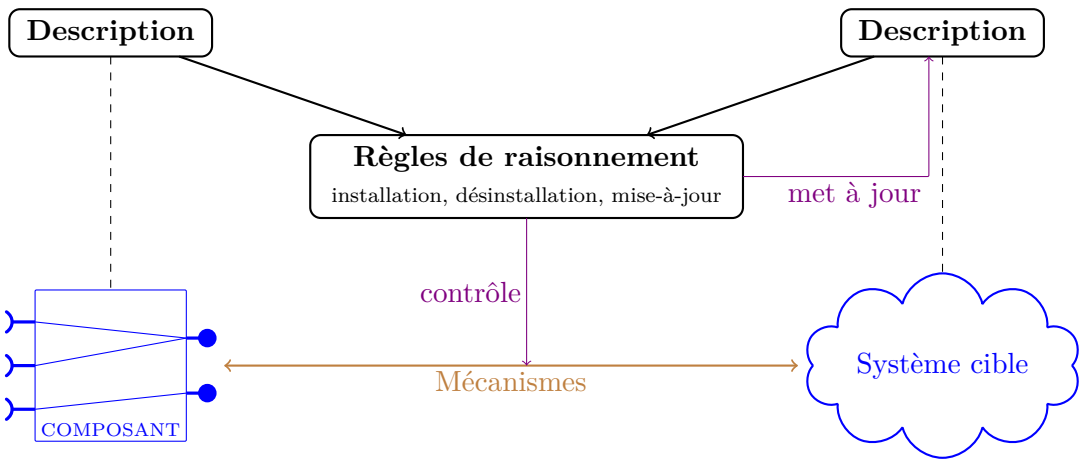


Figure 2.4 – Raisonnement et réalisation du déploiement

3. les services fournis par le composant à désinstaller ne sont pas utilisés par d'autres composants dans le système ;
4. les services fournis par le composant à mettre à jour et utilisés par d'autres composants doivent être fournis également par le nouveau composant.

Pour répondre à ces questions, nous avons besoin de décrire les ressources du déploiement : (1) les dépendances du composant en question (ses contraintes et son effet sur le système cible) et (2) le système cible et son architecture. C'est l'objet de la section suivante que de définir ces éléments formellement.

2.3 Des descriptions mathématiques

Le modèle mathématique proposé repose sur la notion de *dépendance* qui va décrire un composant par l'usage d'un langage *logique*. Un terme de dépendance exprime à la fois les services que le composant peut fournir ainsi que l'ensemble des ressources dont il a besoin pour cela. Le système dans lequel, on déploie est aussi décrit par un terme que l'on appelle le *contexte*.

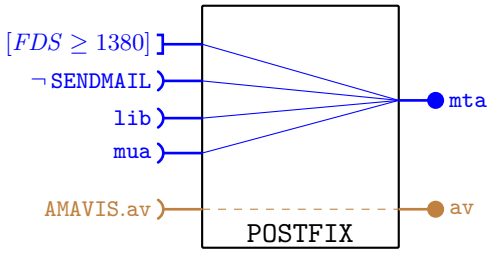
2.3.1 La dépendance

Un composant peut fournir potentiellement un ensemble de services et spécifie pour chacun de ces services fournis l'ensemble de ses exigences. Ces *intra-dépendances*, de la forme $P \Rightarrow s$, expriment la pré-condition P d'installation d'un service s. Les services comme les composants vont être des noms. Pour simplifier la lecture des termes, les composants seront en majuscules et les services en minuscules.

Comme un composant fournit en général plusieurs services, des opérateurs permettent de composer ces dépendances élémentaires. Nous avons retenu une syntaxe visuellement différentes de la logique des prédicats usuelle pour ne pas mélanger les opérateurs de pré-condition avec ceux qui composent des dépendances. Intuitivement, ces opérateurs permettent d'indiquer qu'une dépendance est optionnelle ($?\,D$), qu'elle correspond à la conjonction ($D \bullet D$) ou à la disjonction ($D \# D$) de deux dépendances.

$$
\begin{aligned}
D &::= P \Rightarrow s \mid D \bullet D \mid D \# D \mid ?\,D \\
P &::= \mathit{true} \mid P \wedge P \mid Q \\
Q &::= Q \vee Q \mid [v\ O\ \mathit{val}] \mid \neg\, s \mid \neg\, c \mid c.s \mid s \\
O &::= > \mid \geq \mid < \mid \leq \mid = \mid \neq
\end{aligned}
$$

(a) La grammaire des dépendances



(b) La dépendance de POSTFIX

Figure 2.5 – La notion de dépendance

Les pré-conditions d'installation d'un service sont données sous forme normale conjonctive (*i.e.* comme une conjonction de disjonctions) pour simplifier la description de certaines règles de déploiement. Les exigences primitives sont alors de cinq formes possibles :

1. Une exigence de ressource est traduite par une opération de comparaison entre une *variable* du contexte (le système) et une valeur (entière). Ainsi, par exemple, une exigence d'espace disque disponible pourrait s'écrire $FDS \geq 1380$ si FDS représente l'espace disque disponible [1].
2. On peut interdire la présence d'un composant $\neg\, c$ ou d'un service $\neg\, s$. Dans ce cas, si le composant ou le service est disponible dans le contexte, l'installation n'est pas possible [2].
3. Enfin, le service peut exiger la disponibilité d'un service s fourni par n'importe quel composant ou d'un service fourni par un composant $c.s$.

La grammaire complète du langage est donnée dans la partie droite de la figure 2.5(a). Ainsi, par exemple, les principales dépendances d'un composant `POSTFIX` pourraient être exprimées par le terme ci-dessous de façon à spécifier (1) l'espace disque disponible nécessaire, (2) les librairies requises, (3) le conflit avec le composant `SENDMAIL`, (4) l'exigence du service `mua` pour fournir le service `mta` et (5) la nécessité d'un service `av` d'un composant `AMAVIS` pour fournir l'anti-virus `av`. Notons que la deuxième partie de la dépendance est optionnelle. Ainsi, si l'exigence du service `av` n'est pas satisfaite, le composant pourra quand même être installé mais ne fournira pas le service d'anti-virus. Pour rendre plus visuel ces termes, nous utilisons aussi une représentation graphique telle que présentée dans la figure 2.5(b).

$$(\underbrace{[FDS \geq 1380]}_{(1)} \wedge \underbrace{\texttt{lib}}_{(2)} \wedge \underbrace{\neg\,\texttt{SENDMAIL}}_{(3)} \wedge \underbrace{\texttt{mua}}_{(4)} \Rightarrow \texttt{mta}) \bullet ?\ (\underbrace{\texttt{AMAVIS.av}}_{(5)} \Rightarrow \texttt{av})$$

2.3.2 Le contexte

Les ressources et l'architecture du système cible sont modélisées par le *contexte*. Ce dernier correspond à l'ensemble des dépendances de tous ses composants (les composants installés).

1. Il existe de nombreux travaux de standardisation de langage de description de ressource. Nous n'avons pas abordé cette problématique pendant nos travaux sur le déploiement.

2. Un composant peut interdire un service qu'il fournit ($\ldots \neg\, s \cdots \Rightarrow s$). Cette contrainte peut être utilisée pour interdire à tout autre composant qui fournit s d'être installé. Une telle dépendance n'est pas contradictoire car le prédicat s'applique sur le contexte avant installation et le service est ajouté après installation.

Pour faciliter sa manipulation, ce contexte contient uniquement une approximation sûre de l'information nécessaire pour réaliser un déploiement correct. Ainsi, le contexte est composé de (1) l'ensemble $\mathcal{E}$ des valeurs des variables d'environnement (qui modélise l'état des ressources), (2) l'ensemble $\mathcal{C}$ des couples $(c, \mathcal{P}_s)$ mémorisant, pour chaque composant installé c, l'ensemble de ses services fournis $\mathcal{P}_s$. Chaque service fourni est associé à l'ensemble des services $\mathcal{F}_s$ et celui des composants $\mathcal{F}_c$ qu'il interdit et (3) un *graphe de dépendance* $\mathcal{G}$ pour mémoriser l'ensemble des dépendances potentielles dans le système. Dans le graphe de dépendance, un nœud représente un service disponible ainsi que son fournisseur $(c.s)$. Pour chaque service s_1 de c_1 dont dépend le service s_2 fourni par c_2, un arc (dirigé) *est_requis_par* $c_1.s_1 \overset{x}{\mapsto} c_2.s_2$ existe dans le graphe de dépendance où x indique sa contingence : M pour une dépendance obligatoire ou O pour une dépendance optionnelle. Il est important de noter que, dans notre proposition, le système de décision va conserver tous les liens de dépendances que le mécanisme réalisant le déploiement pourrait potentiellement créer. Ainsi, le graphe de dépendance inclut plus de dépendances qu'il n'y en aura réellement dans le système cible. Il représente donc une approximation sûre de la configuration actuelle du système cible et son architecture en termes de composants et services.

On pourrait, en supportant un dialogue plus important entre le système de vérification du déploiement et la plate-forme réalisant le déploiement, construire un graphe plus précis et donc introduisant moins de dépendances superflues. Mais nous n'avons pas exploré cette piste pour conserver un très faible couplage entre le solveur et l'outil de réalisation du déploiement. Notons que cela permet aussi de ne pas exiger trop de fonctionnalité de l'outil de déploiement (en général, ces outils ne fournissent pas de moyens de savoir exactement les opérations qu'ils ont réalisées).

Le graphe de dépendances est construit pendant l'installation et utilisé pendant la désinstallation. Il est également mis à jour par chaque opération de déploiement. En faisant l'hypothèse (raisonnable) que toute installation est faite par notre outil, le contexte contient bien une approximation sûre de l'état de la machine. On pourrait également imaginer des opérations de re-synchronisation dans le cas où des divergences entre la description et le système cible apparaîtraient.

Remarque : *Par construction, le graphe de dépendance est acyclique car la relation $n_1 \mapsto n_2$ implique que n_1 était disponible avant n_2. Dans la pratique cette façon de construire le système de déploiement peut être une limitation car deux composants peuvent être mutuellement dépendants. Nous défendons l'idée que des composants mutuellement dépendants doivent être empaquetés ensembles au préalable. Dans le manuscrit de la thèse de Meriem Belguidoum [P17], on trouve la définition d'un opérateur de composition permettant la construction d'un assemblage de composants section 5.2.3 pages 108 à 114. Nous offrons ainsi la possibilité d'installer des ensembles de composants exhibant un cycle de dépendance en utilisant notre système moyennant une étape supplémentaire d'empaquetage.*

Pour simplifier la présentation de nos règles de déploiement, nous définissons des fonctions qui calculent l'ensemble des services disponibles AS, celui des composants disponibles AC, celui des services interdits FS et enfin celui des composants interdits FC. Elles sont définies comme

suit [3] :

$$\begin{cases} AS(Ctx) = \{s \mid (-, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s, -, -) \in \mathcal{P}_s\} \\ AC(Ctx) = \{c \mid (c, -) \in Ctx.\mathcal{C}\} \\ FS(Ctx) = \bigcup\{\mathcal{F}_s \mid (-, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (-, \mathcal{F}_s, -) \in \mathcal{P}_s\} \\ FC(Ctx) = \bigcup\{\mathcal{F}_c \mid (-, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (-, -, \mathcal{F}_c) \in \mathcal{P}_s\} \end{cases}$$

Ces descriptions formelles de dépendances des composants ainsi que du contexte permettent de réaliser un déploiement de manière sûre. La section suivante présente les règles pour le réaliser.

2.4 Des règles mathématiques de déploiement

Nous allons présenter, ici, uniquement une partie des formalisations pour illustrer leur forme et les choix réalisés. Plus précisément, nous allons présenter le détail de l'installation, plus rapidement la désinstallation et la mise à jour qui sont des parties un peu plus originales de nos travaux.

Nous rappelons que les opérations de déploiement sont décomposées en deux étapes, on commence par déterminer si l'opération est possible et si c'est le cas, on calcule l'effet de l'opération (et donc le nouveau contexte). Ces règles sont présentées sous la forme d'un ensemble de règles de dérivation (suivant la déduction naturelle). Si une preuve de correction de l'opération est possible suivant cet ensemble de règle dans le contexte courant, l'opération est possible et son effet peut être calculé.

2.4.1 L'installation

L'installation d'un composant dans un système correspond à l'ajout d'un composant dans un environnement. Pour cela, la première étape consiste à vérifier son *installabilité* [4] et la seconde consiste à calculer l'*effet* de son installation. Cet effet décrira les nouveaux ensembles de services et de composants disponibles et interdits, ainsi que le graphe de dépendance.

Installabilité

Avant d'autoriser l'installation d'un composant c de dépendance D dans un contexte Ctx, il faut s'assurer que (1) le composant n'est pas interdit par le système cible, (2) les services qu'il requiert sont disponibles dans le contexte et (3) il ne fournit pas des services interdits par le système cible. Soit, en utilisant les règles de vérification de l'installabilité d'un composant $\vdash_{\text{C}}$ présentées dans la figure 2.6 page suivante :

$$\textsc{CComp} : \frac{\overbrace{Ctx \vdash_{\text{C}} D}^{(2) \text{ et } (3)} \qquad \overbrace{c \notin FC(Ctx)}^{(1)}}{Ctx \vdash \texttt{installable}(c : D)}$$

Le principe des règles est de vérifier que pour chaque dépendance *obligatoire* [5], ses exigences sont réalisées (2) et que le service alors fourni n'est pas interdit (3). Pour une dépendance

3. Le caractère − est utilisé pour un choix libre (une variable non utilisée). Les éléments du contexte sont accédés suivant une notation pointée (à la manière d'attributs d'objets).

4. Terme utilisé pour parler de la possibilité de réaliser une installation sans problème.

5. Dans cette phase, les dépendances optionnelles sont ignorées (COpt) car la non disponibilité de telles dépendances n'empêche pas l'installation du composant.

Évaluation des prédicats :

$$\textsc{PTrue}: Ctx \vdash_{\mathrm{P}} true \qquad \textsc{PAnd}: \frac{Ctx \vdash_{\mathrm{P}} P_1 \qquad Ctx \vdash_{\mathrm{P}} P_2}{Ctx \vdash_{\mathrm{P}} P_1 \wedge P_2} \qquad \textsc{POrL}: \frac{Ctx \vdash_{\mathrm{P}} Q_1}{Ctx \vdash_{\mathrm{P}} Q_1 \vee Q_2}$$

$$\textsc{POrR}: \frac{Ctx \vdash_{\mathrm{P}} Q_2}{Ctx \vdash_{\mathrm{P}} Q_1 \vee Q_2} \qquad \textsc{PVar}: \frac{Ctx.\mathcal{E}(v)\ O\ V}{Ctx \vdash_{\mathrm{P}} [v\ O\ V]} \qquad \textsc{PNotS}: \frac{s \notin AS(Ctx)}{Ctx \vdash_{\mathrm{P}} \neg s} \qquad \textsc{PNotC}: \frac{c \notin AC(Ctx)}{Ctx \vdash_{\mathrm{P}} \neg c}$$

$$\textsc{PServ}: \frac{s \in AS(Ctx)}{Ctx \vdash_{\mathrm{P}} s} \qquad \textsc{PComp}: \frac{(c, \mathcal{P}_s) \in Ctx.\mathcal{C} \qquad (s, -, -) \in \mathcal{P}_s}{Ctx \vdash_{\mathrm{P}} c.s}$$

Règles d'installabilité :

$$\textsc{CTriv}: \frac{Ctx \vdash_{\mathrm{P}} P \qquad s \notin FS(Ctx)}{Ctx \vdash_{\mathrm{C}} P \Rightarrow s} \qquad \textsc{CAnd}: \frac{Ctx \vdash_{\mathrm{C}} D_1 \qquad Ctx \vdash_{\mathrm{C}} D_2}{Ctx \vdash_{\mathrm{C}} D_1 \bullet D_2} \qquad \textsc{COpt}: Ctx \vdash_{\mathrm{C}} ?\, D$$

$$\textsc{COrL}: \frac{Ctx \vdash_{\mathrm{C}} D_1}{Ctx \vdash_{\mathrm{C}} D_1 \# D_2} \qquad \textsc{COrR}: \frac{Ctx \vdash_{\mathrm{C}} D_2}{Ctx \vdash_{\mathrm{C}} D_1 \# D_2}$$

Figure 2.6 – Les règles d'installabilité et d'évaluation des prédicats

simple $P \Rightarrow s$, P doit être vrai et le service s ne doit pas être interdit (CTriv). L'évaluation du prédicat P dans le contexte Ctx est présentée dans la première partie de la figure (les règles $\vdash_{\mathrm{P}}$). Les prédicats ont une sémantique usuelle, les propositions de base correspondant à la disponibilité d'un service éventuellement avec son fournisseur. Les dépendances sont ensuite combinées $_ \bullet _$ ayant un sens de conjonction (CAnd) alors que $_ \# _$ se comporte comme une disjonction (COrL et COrR).

Installation

Une fois l'installabilité prouvée, l'effet de l'installation d'un composant c de dépendance D dans un contexte Ctx doit être calculé. Il y a deux effets possibles : soit $\perp$ si la dépendance n'est pas dérivable, soit un couple contenant les nouveaux services disponibles $\mathcal{P}_s$ (les services effectivement fournis avec leurs nouveaux services et composants interdits $\mathcal{F}_s$ et $\mathcal{F}_c$) ainsi que les nouvelles dépendances (représentées par un graphe de dépendance $\mathcal{G}$). Les effets sont calculés à partir de la dépendance par les règles $\vdash_{\mathrm{I}}$ de la figure 2.7 page suivante :

$$\textsc{IComp}: \frac{Ctx, c \vdash_{\mathrm{I}} D \Rightarrow E}{Ctx \vdash \texttt{install}(c : D) \Rightarrow E}$$

Une fois calculés, les effets sont alors ajoutés au contexte associé au composant (on ajoute $(c, \mathcal{P}_s)$) et le nouveau graphe de dépendance correspond à l'union de l'ancien avec celui de l'effet [6].

6. Il faut noter que le moteur de raisonnement formel ne prend pas en compte la mise à jour des variables d'environnement (espace disque, mémoire, etc.). Cette mise à jour est prise en charge par la plate-forme de déploiement. Leur valeur étant obtenue par des sondes.

$$\text{ITriv}: \frac{Ctx \vdash_{\text{P}} P \qquad s \notin FS(Ctx) \qquad \texttt{forbid}(Ctx, P) = \mathcal{F}_s, \mathcal{F}_c}{Ctx, c \vdash_{\text{I}} P \Rightarrow s \Rightarrow \{(s, \mathcal{F}_s, \mathcal{F}_c)\}, \texttt{graph}(Ctx, c, s, P)}$$

$$\text{INot}_1: \frac{Ctx \not\vdash_{\text{P}} P}{Ctx, c \vdash_{\text{I}} P \Rightarrow s \Rightarrow \bot} \qquad \text{INot}_2: \frac{s \in FS(Ctx)}{Ctx, c \vdash_{\text{I}} P \Rightarrow s \Rightarrow \bot} \qquad \text{IOpt}: \frac{Ctx, c \vdash_{\text{I}} D \Rightarrow E}{Ctx, c \vdash_{\text{I}} ?\, D \Rightarrow ?\, E}$$

$$\text{IAnd}: \frac{Ctx, c \vdash_{\text{I}} D_1 \Rightarrow E_1 \qquad Ctx, c \vdash_{\text{I}} D_2 \Rightarrow E_2}{Ctx, c \vdash_{\text{I}} D_1 \bullet D_2 \Rightarrow E_1 \cup E_2} \qquad \text{IOr}: \frac{Ctx, c \vdash_{\text{I}} D_1 \Rightarrow E_1 \qquad Ctx, c \vdash_{\text{I}} D_2 \Rightarrow E_2}{Ctx, c \vdash_{\text{I}} D_1 \# D_2 \Rightarrow E_1 \oplus E_2}$$

Figure 2.7 – Les règles d'installation

Pour simplifier l'écriture des règles, on utilise une fonction pour calculer les services et les composants interdits, une fonction de calcul du graphe de dépendance et on définit l'optionalisation ?, l'union $\cup$ et le cumul d'effet $\oplus$:

– La fonction `forbid` calcule les services et les composants interdits en collectant les négations introduites dans les prédicats de la dépendance.

$$\begin{cases} \texttt{forbid}(Ctx, \neg s) = \begin{cases} \{s\}, \varnothing \ \texttt{si}\ s \notin AS(Ctx) \\ \varnothing, \varnothing \ \texttt{sinon} \end{cases} \\ \texttt{forbid}(Ctx, \neg c) = \begin{cases} \varnothing, \{c\} \ \texttt{si}\ c \notin AC(Ctx) \\ \varnothing, \varnothing \ \texttt{sinon} \end{cases} \\ \texttt{forbid}(Ctx, true) = \texttt{forbid}(Ctx, [v\ O\ V]) = \varnothing, \varnothing \\ \texttt{forbid}(Ctx, s) = \texttt{forbid}(Ctx, c.s) = \varnothing, \varnothing \\ \texttt{forbid}(Ctx, P_1 \wedge P_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ où } \texttt{forbid}(Ctx, P_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \\ \texttt{forbid}(Ctx, Q_1 \vee Q_2) = \mathcal{F}_s^1 \cup \mathcal{F}_s^2, \mathcal{F}_c^1 \cup \mathcal{F}_c^2 \text{ où } \texttt{forbid}(Ctx, Q_i) = \mathcal{F}_s^i, \mathcal{F}_c^i \end{cases}$$

La seule particularité de cette définition est le cas de la disjonction. En effet, plusieurs sous termes de la disjonction peuvent interdire des services ou des composants. Par exemple, dans l'expression de dépendance $\neg a \vee \neg b \Rightarrow s$, a ou b peuvent être interdits selon le contexte. Ainsi, par exemple, si a est disponible (resp. b) interdire b (resp. a) est suffisant. Si aucun des deux n'est disponible, il serait suffisant de conserver la disjonction qui pourrait être ensuite modifiée (par l'installation de a par exemple). Conserver cette précision requiert un système complexe et coûteux, nous avons donc opté pour un système plus simple qui reste sûr mais est plus contraignant : tous les services non disponibles précédés par une négation sont interdits (c'est donc la même règle que pour la conjonction).

– La fonction `graph` calcule le graphe de dépendance d'un service s produit par un composant c. Le principe de ce calcul est de collecter pour chaque exigence de service s' (règle GServ) ou $c'.s'$ (règle GServC) la dépendance avec le service en cours d'analyse. Notons que dans le cas s' (règle GServ), tous les fournisseurs potentiels de s' sont considérés comme liés, ce qui fournit une approximation sûre. Dans le cas de la disjonction, on retrouve le choix de

cumuler présenté pour `forbid`.

$$\begin{cases} \texttt{graph}(Ctx, c, s, c'.s') = \{c'.s' \overset{\mathsf{M}}{\longmapsto} c.s \mid (c', \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s', -, -) \in \mathcal{P}_s\} \\ \texttt{graph}(Ctx, c, s, s') = \{c'.s' \overset{\mathsf{M}}{\longmapsto} c.s \mid (c', \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s', -, -) \in \mathcal{P}_s\} \\ \texttt{graph}(Ctx, c, s, true) = \texttt{graph}(Ctx, c, s, [v\ O\ V]) = \varnothing \\ \texttt{graph}(Ctx, c, s, \neg s') = \texttt{graph}(Ctx, c, s, \neg c') = \varnothing \\ \texttt{graph}(Ctx, c, s, P_1 \wedge P_2) = \mathcal{G}_1 \cup \mathcal{G}_2 \text{ où } \texttt{graph}(Ctx, c, s, P_i) = \mathcal{G}_i \\ \texttt{graph}(Ctx, c, s, Q_1 \vee Q_2) = \mathcal{G}_1 \cup \mathcal{G}_2 \text{ où } \texttt{graph}(Ctx, c, s, Q_i) = \mathcal{G}_i \end{cases}$$

– Si une dépendance optionnelle n'est pas dérivable alors il n'y a aucun effet. Dans le cas contraire, seul le graphe de dépendance est modifié pour que toutes les dépendances aient une étiquette O.

$$\begin{cases} ?\bot = (\varnothing, \varnothing) \\ ?(\mathcal{P}_s, \mathcal{G}) = (\mathcal{P}_s, \{c.s \overset{\mathsf{O}}{\longmapsto} c'.s' \mid c.s \overset{-}{\longmapsto} c'.s' \in \mathcal{G}\}) \end{cases}$$

Il faut noter que les seules dépendances présentes dans un effet sont de la forme $c.s \overset{-}{\longmapsto} c'.s'$ où c' est le composant en cours d'installation et s' un de ses services. Cette propriété peut être montrée par induction sur la forme des effets.

– Lors d'une conjonction de dépendance $D_1 \bullet D_2$, les deux dépendances doivent être dérivables et l'effet résultant est dans ce cas l'union des différents effets.

$$\begin{cases} \bot \cup E = E \cup \bot = \bot \\ (\mathcal{P}_s^1, \mathcal{G}_1) \cup (\mathcal{P}_s^2, \mathcal{G}_2) = (\mathcal{P}_s^1 \cup \mathcal{P}_s^2, \mathcal{G}_1 \cup \mathcal{G}_2) \end{cases}$$

– Enfin, lors de la disjonction $D_1 \# D_2$ l'effet est soit celui de la première dépendance D_1 si elle est vérifiée, soit celui de D_2 dans le cas contraire.

$$\begin{cases} \bot \oplus E = E \\ (\mathcal{P}_s, \mathcal{G}) \oplus E = (\mathcal{P}_s, \mathcal{G}) \end{cases}$$

Remarque : *La présence des deux systèmes n'est pas absolument nécessaire puisque le calcul d'effet produit $\bot$ lorsque la dépendance n'est pas correcte. Cependant, pour rendre plus simple le traitement, la présentation et optimiser un peu, nous avons choisi de ne calculer l'effet qu'en cas d'installabilité.*

Remarque : *Le choix de baser le calcul d'effet sur la même description de dépendance que celle de la pré-condition d'installation fait que le comportement de la conditionnelle est différent de celui du choix. En effet, si True était défini comme une dépendance toujours vraie ne produisant pas d'effet, on aurait $?D \neq D \# True$. Ces deux termes mènent à la même pré-condition, renvoient la même liste de services installés mais ne renvoie pas le même graphe de dépendance. Dans le premier cas, les dépendances induites par D sont marquées comme optionnelle alors qu'elles ne le sont pas pour le deuxième terme.*

2.4.2 Description de la désinstallation

De manière similaire, la désinstallation d'un composant est réalisée en deux étapes. La première vérifie la faisabilité de la désinstallation en se basant sur le graphe de dépendance.

Cette vérification consiste à s'assurer que les services fournis par le composant en question ne sont pas utilisés par d'autres composants. La deuxième étape consiste à calculer l'effet de la désinstallation sur le contexte, c'est-à-dire la désinstallation effective du composant et de tous ses services fournis au niveau du contexte.

Désinstallabilité

Le principe consiste à parcourir le graphe de dépendance pour s'assurer qu'aucun des services fournis par le composant en question n'est nécessaire à un autre composant. Un première option sûre mais très contraignante est d'autoriser uniquement le retrait d'un service s lorsqu'aucun service fourni par un composant client de s n'est modifié. Dans la pratique cela oblige souvent l'utilisateur à retirer d'autres services d'abord (désinstaller d'autres composants). Dans notre proposition, nous sommes un peu plus lâche et autorisons le retrait en cascade de services (et donc éventuellement la désinstallation de composants) si tous ces services dépendent les uns des autres de manière optionnelle.

Remarque : *Dans les systèmes proposés plus récemment, les composants sont classés en deux catégories : les composants de l'utilisateurs (ceux dont il a explicitement demandé l'installation) et les autres (leur installation est la conséquence d'une demande d'installation d'un autre composant). Ces catégories sont lors utilisées pour contraindre les désinstallations en cascade : un composant* utilisateur *est désinstallé uniquement si l'utilisateur le demande.*

Ainsi, il n'est possible de désinstaller que des composants dont les services ne sont pas utilisés (ses services sont des feuilles du graphe de dépendance) ou ne sont utilisés que de façon optionnelle (tous les chemins sortant du nœud sont des suites d'arcs optionnels). Pour cela, on définit l'ensemble des dépendances obligatoires (MD) d'un service s fourni par un composant c dans un graphe de dépendance $\mathcal{G}$. C'est l'ensemble des services qui le requiert directement ou indirectement de façon obligatoire :

$$MD(\mathcal{G}, c.s) = \{c'.s' \mid c.s \overset{\mathsf{M}}{\mapsto} c'.s' \in \mathcal{G}\} \cup \bigcup \{MD(\mathcal{G}, c'.s') \mid c.s \overset{-}{\mapsto} c'.s' \in \mathcal{G}\}$$

On peut alors définir la désinstallabilité par :

$$\textsc{CDComp} : \frac{(c, \mathcal{P}_s) \in Ctx.\mathcal{C} \qquad \bigcup \{MD(Ctx.\mathcal{G}, c.s) \mid (s, -, -) \in \mathcal{P}_s\} = \varnothing}{Ctx \vdash_{\mathrm{D}} \texttt{deinstallable}(c)}$$

Désinstallation

Le principe de la désinstallation d'un composant c est de supprimer tous les services qu'il fournit dans le graphe de dépendance. Ces services ne sont pas utilisés de manière obligatoire par d'autres composants (assuré par la désinstallabilité). En revanche, des services peuvent en dépendre de manière optionnelle, ils doivent être désinstallés car ils n'ont plus leurs requis. Ce processus doit être appliqué de manière récursive. Enfin, il convient de supprimer tous les composants qui ne fourniraient plus aucun service. Pour effectuer la suppression d'un ensemble de nœuds d'un graphe de dépendance, on définit l'opération suivante :

$$\mathcal{G} \setminus N = \{n_1 \overset{x}{\mapsto} n_2 \mid n_1 \overset{x}{\mapsto} n_2 \in \mathcal{G} \wedge n_1 \notin N \wedge n_2 \notin N\}$$

Il convient également de calculer l'ensemble de dépendances optionnelles (OD) d'un service s fourni par un composant c dans un graphe de dépendance $\mathcal{G}$. C'est l'ensemble des services qui en dépendent directement ou indirectement de façon optionnelle. Il est défini comme suit :

$$OD(\mathcal{G}, c.s) = \bigcup \{\{c'.s'\} \cup OD(\mathcal{G}, c'.s') \mid c.s \overset{\mathsf{O}}{\mapsto} c'.s' \in \mathcal{G}\}$$

On peut alors calculer l'ensemble des nœuds à supprimer du graphe de dépendance :

$$\text{DComp} : \frac{(c, \mathcal{P}_s) \in Ctx.\mathcal{C}}{Ctx \vdash_{\mathrm{E}} \texttt{deinstall}(c) \Rightarrow \bigcup_{(s,-,-) \in \mathcal{P}_s} (\{c.s\} \cup OD(Ctx.\mathcal{G}, c.s))}$$

Une fois cet effet calculé, il faut :

1. supprimer le composant c avec ses services fournis $\mathcal{P}_s$;
2. supprimer les nœuds calculés par la règle DComp du graphe de dépendance ;
3. pour chaque service $c'.s'$ supprimé du graphe qui n'est pas fourni par c ($c' \neq c$), il faut retirer s' des services fournis par c' et si l'ensemble des services fournis par c' devient vide, on supprime également c'.

2.4.3 Mise à jour des composants

Si l'installation et la désinstallation sont des opérations importantes, elles ne sont pas suffisantes. En effet, la mise à jour d'un composant ne consiste pas à désinstaller l'ancien composant et à installer le nouveau. Le problème est que si on désinstallait l'ancien composant, le système serait probablement dans un état incorrect (les composants qui en dépendent n'ont plus leurs pré-requis satisfaits).

Cette problématique a été beaucoup étudié dans le cadre de la substituabilité qui est une propriété essentielle des langages de programmation polymorphe. À l'origine, le principe de substitution est dû à Liskov dans [P73]. Il est fortement lié à la programmation objet et spécifie qu'un type t_1 est un sous-type de t_2 si tout objet de type t_2 peut être remplacé par un objet de type t_1. Ce principe a été repris dans le cadre des composants logiciels ([P100] par exemple). Il est néanmoins contraignant car il impose au nouveau composant de respecter le type du premier ce qui n'est pas toujours possible. Une des difficultés est que ces modèles sont généralement ouverts : ils cherchent à garantir la substituabilité quelque soit le contexte d'exécution. Alors que parfois, en disposant d'un contexte plus restreint (par exemple, je sais que mon logiciel n'utilise pas une certaine fonctionnalité s), on peut remplacer un composant par un autre qui n'est pas son sous-type (il ne fournit pas s par exemple). On trouve dans [P23] une telle notion de substituabilité définie pour des composants boite noire. Le principe est de vérifier que la substitution du composant conserve la consistance de la configuration préalable. Ainsi, en s'inspirant de cette thèse, on peut proposer de définir deux formes de substituabilité d'un composant c_{old} par un composant c_{new} :

1. c_{old} est *substituable strictement* par c_{new} si c_{new} peut remplacer c_{old} dans tous les contextes.
2. c_{old} est *substituable* par c_{new} dans le contexte Ctx si c_{new} peut remplacer c_{old} dans ce contexte.

Pour décider cette substituabilité, il faut comparer les exigences et les fournis du nouveau composant par rapport à ceux de l'ancien. Cette comparaison conduit à seize situations suivant

Exiger \ Fournir	plus	autant	moins	différent
autant/moins	**NC**		**NON**	**NC+NON**
différent/plus	**NR+NC**	**NR**	**NR+NON**	**NR+NC+NON**

TABLE 2.1 – Les condition de substituabilité

que les services fournis (resp. requis) de c_{new} sont inclus dans, sont égaux à, contiennent ou sont différents[7] de ceux fournis (resp. requis) de c_{old}. Une analyse plus précise permet de simplifier et de construire la table 2.1 qui réduit à huit situations et seulement trois conditions de substituabilité :

- **NR**[8] : il faut vérifier les nouvelles exigences de c_{new} ;
- **NC**[9] : il faut vérifier que les nouveaux services fournis par c_{new} ne sont pas en conflit avec le contexte ;
- **NON** : il faut vérifier que les services précédemment fournis par c_{old} et qui ne sont plus fournis par c_{new} ne sont pas utilisés de façon obligatoire dans le contexte.

La substituabilité stricte correspond alors à la case vide (elle n'exige aucune vérification) et correspond au cas où c_{new} exige la même chose ou moins et fournit autant que c_{old}. Les sept autres cas sont des substituabilités qui dépendent du contexte. Ainsi, pour déterminer si un composant peut en remplacer un autre, il faut : (1) être capable de déterminer dans quel cas on se trouve et (2) évaluer les conditions correspondantes **NR**, **NC**, **NON**. La première tâche est simple pour les services fournis, il suffit de cumuler tous les membres droits des dépendances. En revanche, déterminer s'il y a plus ou moins d'exigences est complexe car il s'agit de comparer des prédicats. Ainsi, pour simplifier, nous ignorons la distinction sur les exigences et cumulons les conditions. Les trois conditions peuvent également être réinterprétées en utilisant les opérations de déploiement déjà définies. Ainsi, vérifier **NR** (resp. **NC**) correspond à la vérification des nouvelles exigences (resp. à la vérification des nouveaux conflits). Ces deux propriétés sont impliquées par la condition d'installabilité du nouveau composant. Pour **NON**, il faut s'assurer que chaque service fourni par c_{old} mais plus par c_{new} n'est pas utilisé de manière obligatoire dans le contexte (son ensemble de dépendances obligatoires *MD* est vide). Ce qui correspond à vérifier la désinstallabilité d'une partie de c_{old}. Si l'on note $\texttt{serv}(Ctx, c) = \{s \mid (s, -, -) \in \mathcal{P}_s \wedge (c, \mathcal{P}_s) \in Ctx.\mathcal{C}\}$, l'ensemble des services de c_{old} qui ne sont pas fournis par par c_{new} est $\texttt{serv}(c_{old}) \setminus \texttt{serv}(c_{new})$.

Ainsi, nous utilisons la table 2.2 page ci-contre pour assurer la substituabilité. L'algorithme consiste donc à (1) assurer l'installabilité de c_{new} dans le contexte $Ctx' = Ctx.\mathcal{C} \setminus c_{old}$. Puis si c'est le cas (2) déterminer les services que fournira c_{new} dans Ctx' et les comparer à ceux fournis par c_{old} dans Ctx. Si le fourni est différent ou moindre, il faut alors (3) assurer que les services de c_{old} dans Ctx non fourni par c_{new} ne sont pas nécessaires. Enfin, l'effet de la mise à jour correspondra à la désinstallation des anciens services et à l'installation des nouveaux.

7. Ils peuvent alors éventuellement avoir en commun des services requis (resp. fournis).
8. *new requirement*
9. *new conflict*

Fourni plus	Fourni autant	Fourni moins	Fourni différent
NR+NC	**NR**	**NR+NON**	**NR+NC+NON**
installable(c_{new})			
		serv(c_{old}) \ serv(c_{new}) ne sont pas utilisés	

TABLE 2.2 – Résumé des conditions de substituabilité

2.5 Extension avec des propriétés

Le langage de dépendance et les règles de déploiement ont ensuite été étendus. Pour cela, nous ajoutons la possibilité d'associer des propriétés (non fonctionnelles) aux composants et services ainsi que d'étendre les exigences avec des contraintes sur ces propriétés. Nous n'allons pas décrire en détail ce système et renvoyons le lecteur intéressé à la thèse de Meriem Belguidoum [P17, chapitre 8]. Cette section va simplement en résumer les particularités par rapport au système initial.

2.5.1 Propriétés et contraintes

Dans cette extension, une propriété est caractérisée par un nom et une valeur. Elle peut être associée à un composant ou un service. Les propriétés sont typées. Les types supportés sont les nombres, les chaînes de caractères et les booléens. Un composant (ou un service) peut avoir un ensemble de propriétés. La syntaxe de la définition d'un service ou d'un composant fourni est étendue pour leur ajouter un ensemble de propriétés. Par exemple, un service s avec une propriété p de valeur v est noté $s^{[p=v]}$. Un ensemble de propriétés forme un environnement, noté $\mathcal{E}$. Le *domaine* de cet environnement est l'ensemble des noms des propriétés qu'il contient ($dom([p_1 = v_1, \ldots, p_n = v_n] = \{p_1, \ldots, p_n\}$).

La syntaxe des exigences est étendue pour permettre la spécification des contraintes sur les propriétés des composants ou services requis. Une *contrainte de propriété* φ est définie en utilisant des opérateurs de comparaison notés O pouvant être $>$, $\geq$, $<$, $\leq$, $=$ ou $\neq$. Par exemple, $c^{[\text{version} \geq 3]}.s \Rightarrow s'$ spécifie que pour fournir le service s', on requiert le service s du composant c en version supérieure à 3.

On utilise alors une notion de *satisfiabilité* pour vérifier si un ensemble de propriétés satisfait un ensemble de contraintes. Plus précisément, une contrainte φ est satisfaite par un environnement $\mathcal{E}$ si et seulement si (1) toutes les propriétés de φ ont une valeur dans $\mathcal{E}$ et (2) la valeur d'une propriété dans $\mathcal{E}$ satisfait toutes les contraintes la concernant dans φ. Elle est notée $\varphi \leftarrow \mathcal{E}$ et définie comme suit :

$$\begin{cases} [p_i O_i v_i]_{i \in I} \leftarrow \mathcal{E} = (\{p_i \mid i \in I\} \setminus dom(\mathcal{E}) = \varnothing) \wedge \bigwedge_{i \in I} \mathcal{E}(p_i) O_i v_i \\ [] \leftarrow \mathcal{E} = true \end{cases}$$

Par exemple, si la contrainte est $[version \geq 3]$, elle est satisfaite dans l'environnement $[version = 4]$ car $[version \geq 3] \leftarrow [version = 4] = 4 \geq 3$.

La syntaxe des dépendances est donc modifiée pour que l'on puisse associer :

- des propriétés à un service fourni $P \Rightarrow s^{\mathcal{E}}$, le composant lui-même peut également avoir des propriétés $c^{\mathcal{E}} : D$;

– des contraintes sur les services ou composants requis (s^{φ} et $c^{\varphi_c}.s^{\varphi_s}$) ou interdits ($\neg s^{\varphi}$ et $\neg c^{\varphi}$).

2.5.2 Impacts sur le système

Ajouter la notion de propriété implique que les composants et les services peuvent exister en plusieurs versions suivant leurs propriétés. Il faut donc prendre en compte maintenant une notion d'instance pour les composants et les services. Ainsi, une instance du composant c est identifiée par un couple (c, num) où num est son identifiant entier (calculé lors de l'installation et unique dans un contexte). D'autre part, comme un composant peut fournir plusieurs fois le même service avec des propriétés différentes, il faut identifier ses services. L'identité d'un service, notée id_s, est un quadruplet $(c, num, s, \mathcal{E}_s)$ où le couple (c, num) représente l'identité de son fournisseur, s est son nom et $\mathcal{E}_s$ ses propriétés.

Le contexte a une forme similaire à celle du système initial, il est simplement enrichi de toutes les informations concernant les propriétés et les contraintes. L'ensemble des variables d'environnement n'est pas modifié. L'ensemble des composants est modifié pour que chaque composant mémorise son identité id_c et ses propriétés $\mathcal{E}_c$. Ses éléments sont donc de la forme $(id_c, \mathcal{E}_c, \mathcal{P}_s)$ où $\mathcal{P}_s$ mémorise les services fournis avec leurs propriétés et leurs interdits $(s, \mathcal{E}_s, \mathcal{F}_s, \mathcal{F}_c)$. Ces derniers mémorisent les contraintes associées aux interdictions $((s, \varphi_s)$ et $(c, \varphi_c))$ [10]. Enfin, le graphe de dépendance $\mathcal{G}$ est modifié et ses nœuds sont des identités de service disponible (id_s). Un arc $id_s^1 \xmapsto{x,\varphi_c,\varphi_s} id_s^2$ mémorise en plus les contraintes qui ont été utilisées lors de l'installation pour assurer que le service id_s^1 pouvait satisfaire l'exigence correspondante de id_s^2. Les contraintes sont conservées dans le graphe pour assurer des mises à jour correctes. Par exemple, si un composant c avec la propriété $NS = 3$ est requis par un autre avec une contrainte $NS \geq 3$, c ne peut pas être remplacé par un composant pour qui NS vaut 2.

Toutes les fonctions de calcul des services et composants disponibles ou interdits doivent être adaptées à la nouvelle forme du contexte. L'ajout des propriétés et des contraintes rend plus complexe le processus pour déterminer si un composant ou un service sont disponibles ou interdits. En effet, il faut maintenant utiliser la satisfiabilité de contraintes par des propriétés. Ainsi :

– la fonction `forbid` déjà présentée doit être adaptée :

$$\texttt{forbid}(Ctx, c, \mathcal{E}_c) = \exists (c', \varphi_c) \in FC(Ctx) \mid c' = c \wedge \varphi_c \leftarrow \mathcal{E}_c$$

– la fonction `avail` doit être définie pour indiquer si un service est disponible :

$$\texttt{avail}(Ctx, c.s, \varphi_c, \varphi_s) = \exists (\mathcal{E}_c, \mathcal{E}_s) \mid (c, -, \mathcal{E}_c, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s, \mathcal{E}_s, -, -) \in \mathcal{P}_s \wedge \varphi_c \leftarrow \mathcal{E}_c \wedge \varphi_s \leftarrow \mathcal{E}_s$$

De même, lors du calcul de l'effet d'installation, il faut calculer l'ensemble des services (avec leurs composants fournisseurs) qui vont potentiellement satisfaire une exigence. Il faut pour cela utiliser la satisfiabilité :

$$\begin{cases} Ids(Ctx, c.s, \varphi_c, \varphi_s) = \{(id_c, s, \mathcal{E}_s) | (id_c, \mathcal{E}_c, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s, \mathcal{E}_s, -, -) \in \mathcal{P}_s \wedge \varphi_c \leftarrow \mathcal{E}_c \wedge \varphi_s \leftarrow \mathcal{E}_s\} \\ Ids(Ctx, s, \varphi_s) = \{(id_c, s, \mathcal{E}_s) | (id_c, \mathcal{E}_c, \mathcal{P}_s) \in Ctx.\mathcal{C} \wedge (s, \mathcal{E}_s, -, -) \in \mathcal{P}_s \wedge \varphi_s \leftarrow \mathcal{E}_s\} \end{cases}$$

10. De façon à pouvoir trouver d'autres composants qui pourraient satisfaire ces contraintes lors de changement dans le contexte.

Ainsi l'installabilité suit les mêmes règles que celles de la section précédente mais en utilisant les fonctions `avail` et `forbid` (règles PNotS, PNotC, PServ, PComp, CTriv et CComp) pour vérifier la disponibilité de services et composants qui satisfont les contraintes exigées. De même, l'installation est également similaire sauf pour la production du graphe dont les nœuds sont maintenant des identifiants de services et dont les arcs mémorisent les contraintes utilisées lors de l'installation. Il faut également rappeler que c'est l'installation qui calcule l'identifiant d'un nouveau composant lorsque celui-ci est déclaré installable.

Enfin, désinstallabilité, désinstallation et mise à jour sont très proches également. Il suffit d'utiliser correctement les (*instances* des) composants et des services à la place de leurs noms. Il faut noter que c'est pour la mise à jour que les contraintes sont conservées pour chaque dépendance du graphe. En effet, pour décider si un composant C_{old} peut être remplacé par un composant C_{new}, il faut s'assurer que les services qui sont fournis par C_{new} avec leurs propriétés vérifient toujours les contraintes exigées par les autres composants qui les utilisent (qui sont donc conservées dans le graphe de dépendance).

Dans cette section, nous avons montré comment intégrer un mécanisme de propriétés et contraintes dans notre système de déploiement. Il faut noter que cette extension n'est pas que syntaxique puisqu'elle doit aussi prendre en compte les notions d'instance de service et composant ainsi que la satisfiabilité de contraintes par un ensemble de propriétés (avec leurs valeurs).

2.6 Preuve de correction

Dans le cadre de sa thèse Meriem Belguidoum a produit une preuve manuelle de la correction du système de déploiement sans les propriétés. Un début de mécanisation de cette preuve a été réalisé mais n'a pas pu être mené à son terme par manque de temps. Toutefois, la preuve manuelle semble offrir un niveau de confiance intéressant dans la correction de notre gestion des dépendances lors du déploiement. Cette preuve procède par induction sur la forme des preuves d'installation ou de désinstallation.

Pour être un peu plus précis, les deux propriétés auxquelles nous nous sommes intéressés sont la *réussite* et la *sûreté*. La propriété de *réussite* permet d'assurer que l'application est déployée correctement et donc fonctionnera après son installation (si elle est correcte et la spécification de sa dépendance est correcte aussi [11]). La propriété de *sûreté* garantit qu'une application déployée ne perturbera pas, par effet de bord de son installation, les applications déjà installées. Ces propriétés sont définies dans [P94] pour l'étape de l'installation en prenant en compte des notions de politique et compatibilité. Dans notre cadre, nous adaptons ces propriétés sans utiliser la notion de politique d'installation. Nous avons ainsi défini ces propriétés de la façon suivante :

– La *réussite* de l'installation garantit qu'après son installation un composant c (de dépendance D) a toutes ses exigences (obligatoires) satisfaites et fournit bien tous ces services (obligatoires) dans le nouveau contexte :

11. Ces corrections n'ont pas fait partie de nos travaux car nous considérons qu'elles sont en dehors du déploiement. La correction des dépendances est néanmoins un problème complexe peu abordé et qui mériterait des travaux de recherche pour proposer des théories, méthodes et outils aux développeurs d'application afin de les aider dans la production de ces dépendances.

1. $\forall s \in \texttt{serv}(Ctx_{new}, c), Ctx_{new} \setminus c \vdash_{\mathrm{P}} \texttt{requires}(s, D)$ où la fonction `requires` calcule l'ensemble des exigences d'un service fourni en collectant tous les prédicats pré-conditions de s.
2. $\texttt{SPFO}(D) \subseteq \texttt{serv}(Ctx_{new}, c)$ où `SPFO` est « l'ensemble » des services potentiellement fournis de façon obligatoire. Il s'agit d'un ensemble d'ensembles de configurations potentielles noté sous forme de somme, l'inclusion étant vérifiée par $(A + B) \subseteq C \Leftrightarrow A \subseteq C \vee B \subseteq C$. La potentialité provient de l'opérateur $\#$, en effet, $P_1 \Rightarrow s_1 \# P_2 \Rightarrow s_2$ conduit à $\{s_1\} + \{s_2\}$ (soit on est dans la configuration où on fournit s_1, soit on est dans celle où on fournit s_2).

– La *réussite* de la désinstallation assure que le composant est bien désinstallé (avec ses informations). La partie du graphe de dépendance supprimée est celle qui contient tous les nœuds qu'il fournit.
– La *sûreté* de l'installation assure que tous les composants installés avant l'opération ne sont pas perturbés par sa réalisation. Ils doivent être encore installables dans le nouveau contexte : $\forall c \in Ctx_{old}.\mathcal{C}, Ctx_{new} \setminus c \vdash \texttt{installable}(c : D)$.
– La *sûreté* de la désinstallation d'un composant c_r affirme également que tous les anciens composants restent installables mais aussi que les services obligatoires qui étaient fournis le sont encore et enfin que le graphe n'est pas modifié en terme de contingence :

1. Tous les autres composants sont installables :

$$\forall c \in Ctx_{old}.\mathcal{C} \setminus \{c_r\}, Ctx_{new} \setminus c \vdash \texttt{installable}(c : D)$$

2. Tous les services obligatoires sont encore fournis :

$$\forall c \in Ctx_{old}.\mathcal{C} \setminus \{c_r\}, \texttt{SPFO}(D_c) \subseteq c.\mathcal{P}_s^{Ctx_{new}}$$

3. Tous les services obligatoires du précédent graphe de dépendance qui ne sont pas des services fournis par c_r sont toujours obligatoires dans le nouveau graphe :

$$\forall c \in Ctx_{old}.\mathcal{C} \setminus \{c_r\}, s' \notin c_r.\mathcal{P}_s \wedge s \overset{\mathsf{M}}{\longmapsto} s' \in Ctx_{old}.\mathcal{G} \Rightarrow s \overset{\mathsf{M}}{\longmapsto} s' \in Ctx_{new}.\mathcal{G}$$

2.7 Discussion et perspectives

Les travaux menés et rapportés ici se sont déroulés en parallèle de deux autres initiatives avec lesquelles nous avons eu des échanges.

Les travaux du laboratoire Preuves, Programmes et Systèmes de l'université Paris 7 Diderot menés dans le cadre des projets EDOS puis Mancoosi reposent sur une vision un peu différente. En effet, plutôt que de proposer des règles spécifiques à la gestion des dépendances, ils ont choisi de traduire les dépendances en contraintes booléennes de façon à pouvoir réutiliser des solveurs standards. Ces différents projets ont mené à la proposition d'un format standard de dépendance CUDF [P111] ainsi qu'à une compétition de solveurs pour des problèmes exprimés en CUDF [12]. En se basant sur CUDF et sur des solveurs SAT, le projet a abouti à une proposition d'architecture générale pour la gestion des paquets logiciels [P3]. Notons que pour la gestion des mises à jour complexes [P30], il repose sur une approche similaire à la notre. En effet, ils

12. http://www.mancoosi.org/misc

utilisent une description abstraite du système pour raisonner avant de lancer la mise à jour réellement dans l'outil chargé de réaliser concrètement la mise à jour.

La seconde initiative concerne la gestion des *plugins* d'Eclipse menée par Daniel Le Berre de l'université d'Artois [P70]. Suivant une approche similaire à celle de EDOS, les dépendances sont traduites en contraintes booléennes qui sont ensuite résolues par des solveurs spécifiques. Cela a amené à la production de `sat4j` un solveur SAT écrit en Java [13] [P71].

L'avantage clair des deux approches présentées ci-dessus est de pouvoir réutiliser des solveurs puissants qui ont fait leur preuve. Par contre, un échec de résolution est souvent plus complexe à expliquer à l'utilisateur. Nous avions adopté une approche plus ad-hoc pour deux raisons. Tout d'abord, cela facilite le diagnostique d'échec. Ensuite, car il nous semblait, comme présenté en début de ce chapitre, que le problème de gestion des composants était un peu plus complexe.

Il est à noter que nous n'avons pas exploré l'extension de notre approche à de vraies signatures de services avec des relations plus complexes entre les composants. En effet, usuellement les composants logiciels fournissent ou requièrent des services sur la base d'un contrat de service indiquant par exemple les méthodes que fournit le service. Il semble néanmoins que les travaux réalisés pour l'extension du système à des propriétés non fonctionnelles en section 2.5 page 19 permettent d'entrevoir à travers la notion de contrainte et de satisfiabilité un moyen d'étendre notre système. Cette extension parait tout de même un peu complexe car les règles actuelles qui sont déjà complexes le deviendraient plus encore. Pour conserver un minimum de garanties dans le cadre de cette approche, il faudrait alors sans doute construire simultanément le système de règles et sa preuve dans un assistant à la preuve comme Coq. En effet, la preuve manuelle d'un tel système semble difficile à faire.

Il pourrait également être intéressant de s'assurer que notre système pourrait être traduit vers CUDF et éventuellement de l'étendre si des notions manquent.

Il est à noter que le langage de dépendance présenté ici est relativement pauvre et de nombreuses extensions ont été envisagées mais non expérimentées par manque de temps :

- La notion de dépendance optionnelle indique qu'un service peut ne pas être installé si ses exigences ne sont pas satisfaites mais si elles le sont, il est forcément installé. Pour permettre des installations un peu plus adaptées aux besoins des utilisateurs, il pourrait être bon de disposer également de la possibilité d'indiquer si oui ou non un service optionnel doit être installé (lorsque ces exigences sont satisfaites). Ainsi, un utilisateur qui n'utilise pas un certain service pourrait ne pas avoir à l'installer. On entend souvent l'argument du prix faible de l'espace disque vis-à-vis de ce genre d'optimisation. Mais, au-delà de la question de l'espace, la multiplication de services pose d'autres problèmes plus sensibles : le graphe de dépendance devient plus complexe et rend donc les opérations de déploiement de plus en plus complexes. On peut également mettre en avant le problème de la gestion de la sécurité. En effet, plus un utilisateur a de composants et de services installés plus la probabilité de présence d'une faille de sécurité est importante. Dans ce cadre, diminuer le nombre de services pour se concentrer sur ceux dont l'utilisateur a besoin ne peut être que bénéfique.
- Dans un cadre, un peu similaire, nous avons eu des premières reflexions sur l'ajout d'une notion de politique de déploiement. Une politique de déploiement est, dans notre contexte, un ensemble : (a) de règles de réécriture qui transforme la dépendance d'un composant en une autre dépendance qui sera utilisée pour le déploiement et (b) de contraintes ajoutées au contexte. Ainsi, par exemple, un utilisateur peut vouloir rendre obligatoire un service car il en a besoin. Il peut également vouloir ne pas mettre à jour un service même si de nouvelles

13. http://www.sat4j.org

version existent [14] ou alors ne se mettre à jour que lors de changement majeur de version. Dans son contexte, il peut également vouloir interdire un certain composant (ou une de ses versions) qu'il sait potentiellement dangereux ou qu'il ne souhaite pas installer.

Ces premières reflexions ont conduit à une constatation : il existe plusieurs (forme de) dépendances. Il y a les dépendances qui décrivent la réalisation du composant, ses exigences de fonctionnement ainsi que les services qu'il peut potentiellement fournir. Les dépendances qui sont affichées par les différents intermédiaires [15] entre le producteur du logiciel et le système de déploiement du client final. Ces différentes dépendances doivent bien sûr respecter un ordre pour que le système de déploiement puisse prendre des décisions correctes. Ainsi, si on a un composant c et deux dépendances D_1 et D_2 de ce composant, on note $D_1 \succ D_2$ la relation qui indique que D_1 raffine D_2. Raffiner signifie, ici, que toute opération de déploiement faite avec D_1 reste correcte par rapport à ce qui aurait été fait avec D_2 et a un effet moindre. Plus précisément :

$$(\forall Ctx, Ctx \vdash \texttt{installable}(c : D_1) \Rightarrow Ctx \vdash \texttt{installable}(c : D_2)) \wedge$$
$$(\forall Ctx, Ctx \vdash \texttt{install}(c : D_i) \Rightarrow E_i, E_1 \subset E_2)$$

– Le langage proposé a été testé sur l'écriture des dépendances d'un certain nombre de logiciels. Cette étude a clairement indiqué qu'il faudrait faire un effort sur la proposition d'un langage qui serait plus facile à utiliser à la fois par le développeur de logiciel et par leurs utilisateurs. Il faudrait permettre la construction de dépendances plus modulaires et sans doute mieux distinguer les services fournis des pré-conditions.

Enfin, la notion de graphe de dépendance qui modélise un système me semble particulièrement intéressante. Il conviendrait de se poser la question de fournir des outils pour sa visualisation, la navigation en son sein et même traduire les opérations de déploiement en opérations de réécriture de graphe. Au dela de l'aspect graphique plus intuitif, les différentes mesures de graphe existantes permettraient sans doute de fournir des mesures d'évaluation de l'état d'un système. On pourrait par exemple, imaginer fournir une mesure de la difficulté de mise à jour d'un composant en fonction de sa centralité dans le graphe. Ces approches ne seraient évidemment pas aussi faciles à mettre en place dans une approche de type transformation en problème SAT.

Pour terminer et faire un peu le lien avec le chapitre suivant, mes travaux sur le déploiement m'amène à penser que :

– les dépendances internes sont importantes pour permettre l'automatisation du déploiement, dans ce cadre les composants ne peuvent pas être complètement opaques (ils sont forcément *gris*) ;
– la distinction entre instance et type est nécessaire, le type doit décrire la famille mais c'est l'instance, responsable de fournir le service, qui est responsable de la qualité de service.

14. Dans les outils récent, on parle de *pin*.
15. On peut citer l'entreprise qui vend le composant, la plate-frome de diffusion, l'administrateur du site du client final, le client final lui-même ...

Chapitre 3

La notion de composant logiciel

Ce chapitre synthétise les travaux de recherche menés entre 2007 et 2011. Ces travaux issus de reflexions nées dans l'étude du déploiement ont été réalisés en partie lors d'un séjour d'étude en 2009. Pendant ce séjour, j'ai travaillé trois mois et demi chez Astrium à Toulouse puis trois mois et demi chez Thalès Alenia Space à Cannes. Au sein des ces entreprises, j'ai travaillé à clarifier la notion de composant et à analyser en détail son intégration dans des processus complexes de développement logiciel comme ceux pour réaliser les logiciels de contrôle des satellites. Ces travaux ont donné lieu à la production de documents pour les entreprises d'accueil qui sont confidentiels et ont donc été peu publiés. Ces rapports et donc mon travail dans ces entreprises ont porté sur l'intégration de l'approche composant dans leur processus de développement et ceci dans un cadre d'ingénierie des modèles. Seule une partie sur la notion de composant sensible au contexte a été publiée [34]. Cet article synthétise des expériences sur l'ajout aux composants Fractal d'une description d'observation permettant de les brancher automatiquement sur un système d'observation de l'environnement d'exécution.

Les principales contributions de cette activité de recherche sont :

- Un exposé synthétique sur les notions de composant, connecteur et architecture. Cet exposé permet de mieux comprendre la diversité des modèles de composant.
- Un modèle de composant multi-phase qui clarifie les éléments caractéristiques et les relations pour les composants, connecteurs et architectures. Ce modèle prend le contre-pied des principaux modèles actuels, qui cherchent à rester simples et unitaires, en visant plutôt une certaine complétude.

3.1 La notion de composant

La notion de composant est considérée comme une notion particulièrement prometteuse dans le cadre du développement de logiciel par réutilisation et assemblage d'éléments existants. Dans les premières phases d'un développement, les architectes logiciels utilisent des concepts très abstraits comme les composants et les connecteurs en ignorant les problèmes liés à l'infrastructure d'exécution sous-jacente qui sera nécessaire. Ensuite, quand l'infrastructure est définie, ces concepts abstraits sont réalisés sous forme de code. Cette séparation claire entre l'architecture dite fonctionnelle et la gestion des interactions avec les ressources physiques permet de construire des éléments fonctionnels réutilisables (aussi bien au niveau abstrait qu'au niveau du code). De plus, l'utilisation de langages dédiés pour construire le code en charge de l'interaction avec la

plate-forme physique associée à des techniques de génération simplifie grandement la production de ce code technique. Le coût de développement en est donc fortement réduit.

Il existe de nombreux *modèles de composant* qui reposent sur des concepts et des règles d'usage différents. Beaucoup de ces modèles sont disponibles, ils visent des utilisations générales ou alors sont destinés à des usages spécifiques. Ils vont du modèle simple au modèle complet incluant une infrastructure d'exécution. De plus, l'approche à base de composant est utilisée dans les différentes étapes du cycle de développement d'un logiciel. La diversité des notions et artefacts nécessaires pour couvrir ces différentes étapes depuis la conception jusqu'à la production d'un logiciel exécutable contribue grandement à la complexité du domaine. En effet, une telle variété de concepts et la terminologie rendent impossible la prise en main rapide d'un modèle. Ce qui rend leur compréhension et leur comparaison difficile. D'autant plus que porter une architecture ou un code réalisé pour un modèle dans un autre modèle est très difficile et requiert un très haut niveau d'expertise.

La présentation de la notion de composant qui suit est basée sur l'étude et la synthèse des modèles de composant suivants : ceux d'objet généraux comme les EJB [P89], CCM [P85], OpenCOM [P32] ou SOFA [P25] mais aussi ceux spécifiques à un domaine comme Koala [P88], ECF [P99], AADL [P39] ou PECOS [P44]. Les concepts qu'ils contiennent peuvent être basiques comme la notion de composant d'UML 2.0 [P50] et celle d'OSGi [P90] ou très sophistiqués et utiliser un grand nombre de concepts comme Fractal [P24] ou CCM. Certains modèles de composant comme Wright [P7] ou ACME [P43] et plus généralement ceux de la communauté des langages de description d'architecture (les ADL) couvrent seulement les notions liées à la spécification d'architecture logicielle. Enfin, d'autres comme ArchJava [P6] sont plus centrés autour de la programmation de composants. On trouve quelques articles synthèses comme par exemple [P69] et on peut enfin consulter les ouvrages [P108] et [P93] qui contiennent des synthèse du domaine.

3.2 Généralités

Les composants logiciels ont été proposés comme une solution pour augmenter la productivité logicielle en favorisant la réutilisation à large échelle. Pour rendre la réutilisation possible, un composant doit avoir un faible couplage vis-à-vis de son environnement. Pour cela, l'architecture est l'artefact central du cycle de vie du développement. Par rapport au modèle objet, les composants favorisent la réutilisation en remplaçant l'héritage et les références par la composition et les ports et en raffinant les contrats pour y ajouter les requis. La paradigme est basé sur deux concepts principaux :

- les *éléments architecturaux* qui encapsulent les traitements. On les appelle parfois les briques de base (*building blocks*) (BB). Les fonctionnalités réalisées par ces traitements sont appelés des *services*.
- les *architectures* décrivent (statiquement) l'ensemble des éléments architecturaux ainsi que leurs relations. La *composition* est l'action consistant à mettre en relation les éléments architecturaux.

Il n'existe aucune définition d'élément architectural qui ne soit acceptée universellement [1] mais il y a un certain consensus pour le définir comme un *bout de logiciel* qui possède les propriétés

1. C. Szyperski dans [P108] est très souvent cité à la fois par ses supporteurs mais aussi par ses détracteurs.

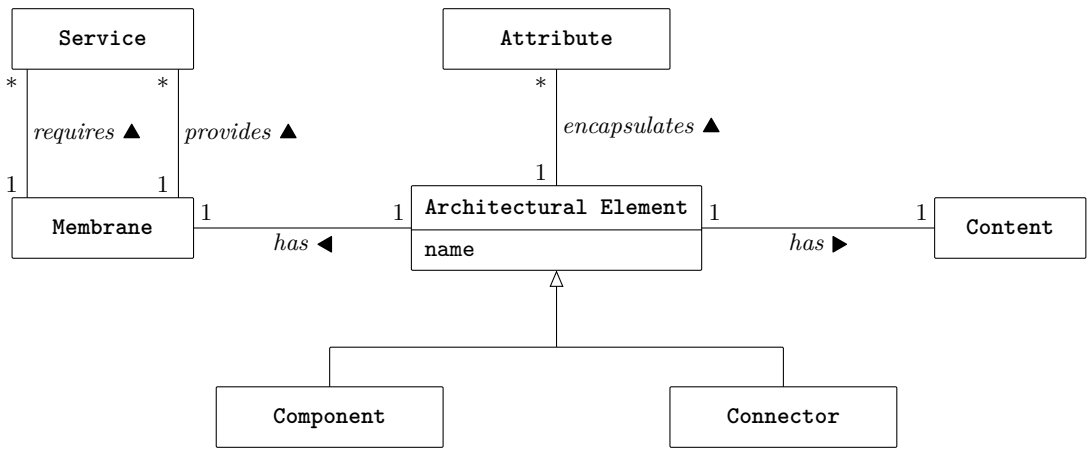


Figure 3.1 – Les éléments architecturaux

suivantes : (1) il est réutilisable, (2) il peut être le sujet d'une composition et (3) il peut être configuré. Les conséquences de ces propriétés sont discutées en détail dans la suite.

3.2.1 Les éléments architecturaux

Un élément architectural (AE), comme décrit dans le méta-modèle de la figure 3.1, est une unité d'encapsulation avec un *contenu* caché pour l'environnement [2] et une *membrane*. Cette membrane est la réification de la frontière de l'AE, son rôle est de gérer les interactions entre le contenu et l'environnement. Ainsi, un AE est une unité de réutilisation.

Les éléments architecturaux sont nommés pour pouvoir être accédés et manipulés. Pour éviter les conflits de noms entre composants, les modèles de composants supposent, en général que tous les composants (connus) aient des noms différents. Bien que non réaliste dans l'absolu, cette hypothèse peut être acceptable dans le cadre d'un producteur de composant. En effet, il connaît tous les composants qu'il produit et peut donc assurer cette unicité. Ici, nous suivons cette règle d'unicité du nom.

Pour être réutilisable, un nom n'est pas suffisant, l'AE doit aussi fournir une description des services qu'il offre et des ressources qu'il requiert. Cette description peut prendre des formes très diverses, depuis une documentation sous forme textuelle jusqu'à une documentation formelle détaillée. Ici, nous utilisons la notion de contrat maintenant très répandue dans le domaine du génie logiciel (voir par exemple [P80]). Ainsi, un AE fournit et requiert un ensemble de [3] services spécifiés par des contrats.

Les éléments architecturaux peuvent être configurés à travers un ensemble d'attributs. Le contenu est paramétré par ces attributs qui sont des données que l'on peut initialiser par une spécification séparée à l'exécution (un fichier de configuration par exemple). Certains modèles

2. Très peu de modèles de composant essayent d'assurer réellement cette encapsulation du contenu. En général, c'est une règle de bonne pratique du développement mais aucun outil ne permet de la vérifier. En effet, dans la plupart des plates-formes d'exécution de composant, un AE peut obtenir une référence ou un pointeur sur le contenu d'un autre composant. Le travail d'Aldrich sur ArchJava aborde cette problématique à travers la proposition d'un système de type qui va assurer une propriété appelée l'intégrité de communication (*communication integrity*) dans [P5].

3. Il est d'usage de pouvoir structurer les contrats d'interaction au sein de plusieurs sous-contrats. Cela permet une bonne séparation des préoccupations lors de la définition et de l'utilisation des contrats d'interaction.

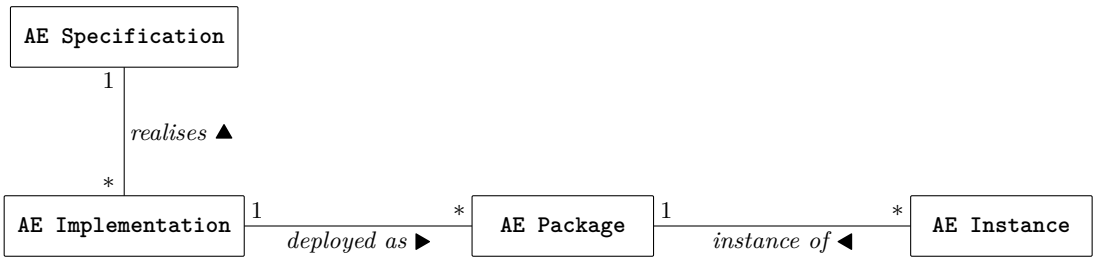


Figure 3.2 – Les différents variations de la notion d'élément architectural

rendent publics ces attributs qui donc peuvent être accédés depuis l'environnement. Dans ce cas, la façon de les accéder ou de les modifier doit faire l'objet d'un service offert.

Enfin, usuellement il y a deux formes d'élément architecturaux : les *composants* et les *connecteurs*. Intuitivement, les composants fournissent les services métiers et les connecteurs prennent en charge le contrôle et la communication entre les composants. Ces deux formes sont discutées en détail plus loin (respectivement sections 3.3 page 32 et 3.4 page 36).

3.2.2 Variabilité au cours du processus de développement

Pendant le processus de développement, la forme d'un AE change. Cheesman et Daniels ont proposé dans [P27] [4] une décomposition, maintenant largement acceptée, des préoccupations concernant les éléments architecturaux. Ainsi, comme montré dans la figure 3.2, nous séparons clairement les différents avatars représentant un AE au cours des phases de développement. Une *spécification* d'AE peut être réalisée par plusieurs *implantations*. Chacune des ces implantations peut être déployée comme un ensemble de *paquets* qui sont ensuite utilisés pour produire les *instances*. Les sections qui décriront les composants et les connecteurs suivront cette structure.

Ces différentes formes sont discutées ci-après.

Spécification La première étape d'un développement consiste à spécifier un élément architectural suivant le méta-modèle de la figure 3.3 page ci-contre. Cette phase doit spécifier les attributs, le contenu et la membrane de l'élément :

- Chaque attribut a un nom unique et un type [5]. Les attributs sont des propriétés configurables au moment de l'instanciation d'un AE. Si ces attributs sont utilisés à l'exécution par l'environnement, il faut définir les services qui permettent cette interaction.
- Spécifier la membrane correspond à la spécification des contrats d'interaction entre l'AE et son environnement. Il y a trois formes de services :
  - les *services fonctionnels* qui peuvent être requis ou fournis. Ces services forment le *type* de l'AE. Attention à ne pas confondre la spécification (que l'implantation doit réaliser) et le type (que les autres composants doivent respecter). Ici la spécification inclut le type.
  - les *services de contrôle fournis* qui peuvent être utilisés par l'environnement pour contrôler l'AE (par exemple, pour le démarrer ou l'arrêter).

4. Ils ne traitent que de la notion de composant mais une généralisation simple est faite ici pour appliquer leurs travaux aux connecteurs.

5. La définition des types est laissée ici imprécise car sa définition n'a que très peu d'impact sur le modèle de composant.

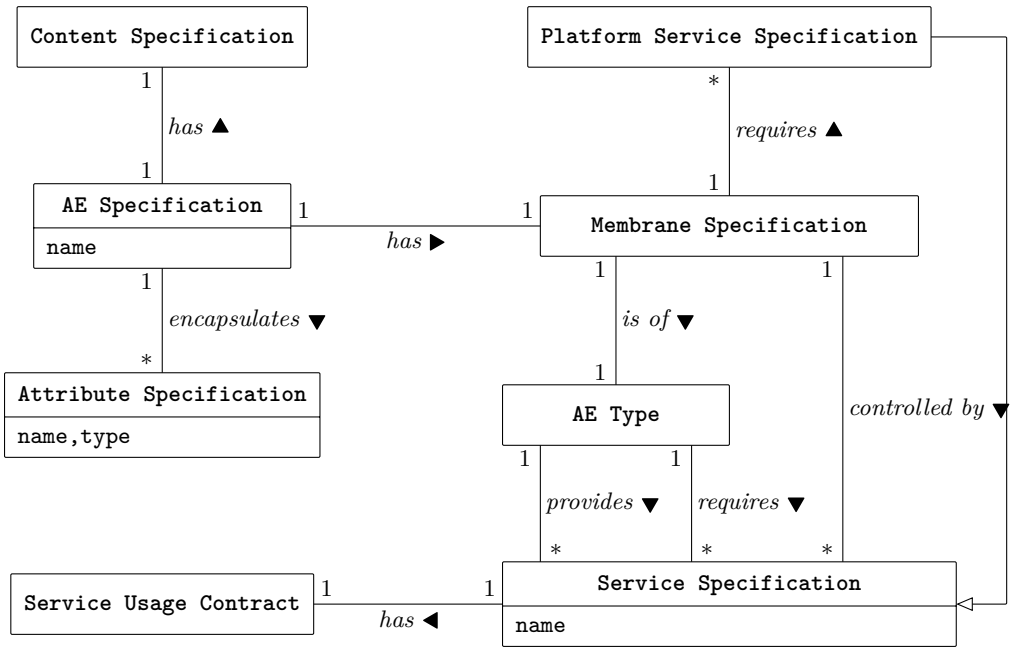


Figure 3.3 – La spécification d'élément architectural

– les *services plates-formes* qui peuvent avoir de nombreuses formes (par exemple, un requis de plate-forme ou un besoin de service d'observation), ne seront pas détaillés ici.

Les deux dernières formes de services sont parfois qualifiées de non-fonctionnelles.

Notons qu'un modèle conforme au méta-modèle de spécification correspond au résultat de la phase de spécification. Des modèles intermédiaires sont sans doute nécessaires. Par exemple, dans une approche de développement orienté modèle (*Model Driven Development, MDD*), certains modèles sont indépendants de la plate-forme (*Platform Independent Models, PIM*) alors que d'autres sont spécifiques de la plate-forme (*Platform Specific Models, PSM*). Il aurait été possible d'intégrer cette évolution au méta-modèle au prix d'une complexité jugée trop grande pour l'objectif de ce document

Implantation Cette phase a pour objectif de définir comment l'AE va se comporter. Cette réalisation suit la structure de spécification avec une réalisation du contenu et une de la membrane [6]. Cette phase suit un processus de développement classique avec des spécifications détaillées puis un code (possiblement partiellement généré). Le méta-modèle de la figure 3.4 page suivante définit tous les concepts de cette phase du développement.

Le contenu peut être primitif ou composite [7] :

6. Souvent un attribut est simplement une donnée stockée dans le composant, sa réalisation ne contient donc pas de comportement et n'apparait donc pas dans ce méta-modèle. Dans certains cas, un attribut est plutôt un couple de fonctions accesseur / modifieur (par exemple pour un attribut stocké de manière transparente dans une base de donnée). Il faut alors réifier ce comportement également dans la membrane.

7. Maintenant, la plupart des modèles de composant permettent la définition de composant à partir d'autres composants.

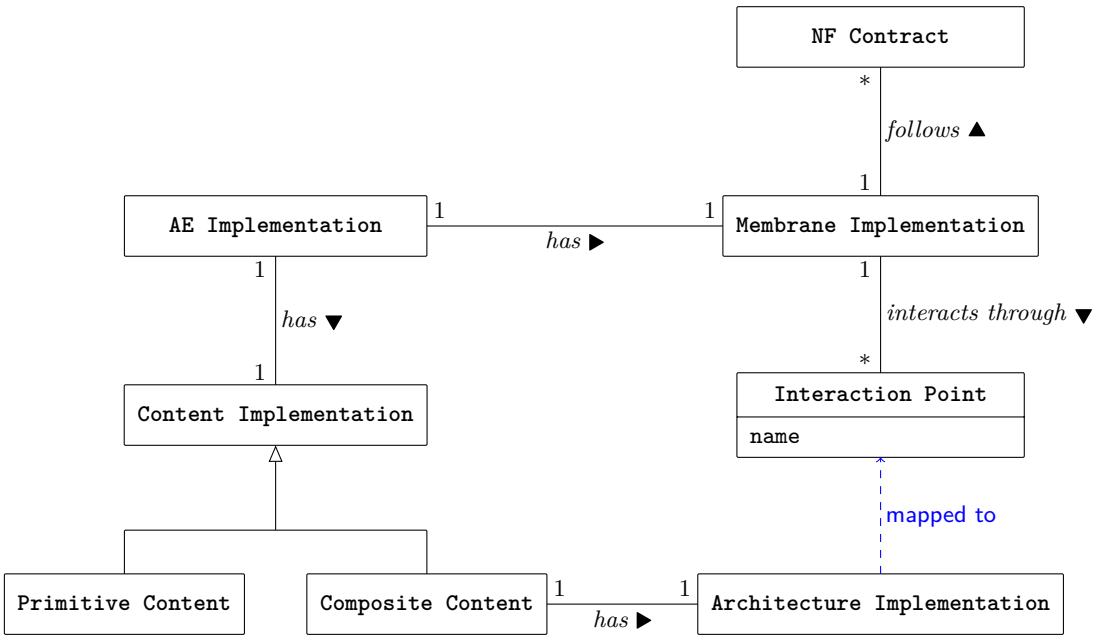


Figure 3.4 – La réalisation d'élément architectural

- Un AE primitif est une boîte noire, sa réalisation est cachée. En général, c'est du code usuel écrit dans un langage comme C / C++ ou Java et suit donc une méthode de réalisation classique.
- Un composite présente une structure interne qui peut être manipulée par l'infrastructure d'exécution. Cette structure est définie sous la forme d'une architecture (du même modèle de composant). Les architectures sont présentées en détail dans la suite, dans la section 3.5 page 39.

Dans le méta-modèle, la relation mapped to entre l'architecture et un point d'interaction représente le fait que les points d'interaction d'un composite peuvent être connectés directement à un de ses sous-éléments. En Fractal, par exemple, il s'agit de la notion de lien (*binding*) importé ou exporté. Dans notre modèle, un composite peut contenir du code et donc supporter des connexions arbitrairement complexes.

La réalisation de la membrane [8] consiste à définir puis réaliser les interactions entre l'AE et son environnement :

- la spécification des services sous forme de points d'interaction,
- les services non-fonctionnels sous la forme de contrats NF qui peuvent prendre trois formes :
  1. des *propriétés NF*, comme la consommation mémoire ou CPU ou un pire temps d'exécution (*Worst Case Execution Time, WCET*),
  2. des *exigences de ressources* comme la forme de l'OS hôte ou les besoins en connectivité,
  3. des configurations de *services plates-formes*, comme par exemple l'utilisation de persistance ou de service de sécurité. Dans la plupart des modèles de composant, ces services sont offerts par la plate-forme d'exécution. Le développeur doit alors seulement fournir

8. Dans certains modèles de composant comme CCM ou EJB, la réalisation de la membrane est appelée un *conteneur*.

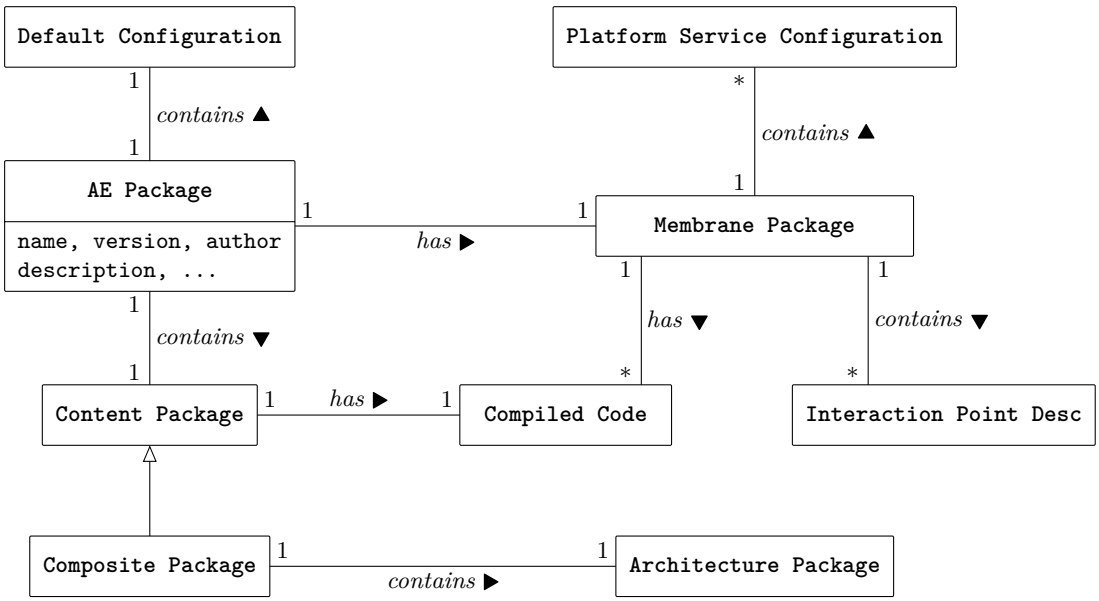


Figure 3.5 – Un paquet d'élément architectural

les paramètres de configuration pour les utiliser. Ainsi, en CCM ou EJB, ceci est fait par un ensemble de descripteurs XML appelés les descripteurs de déploiement.

Dans la plupart des plates-formes existantes, le code de la membrane est généré à partir d'une description déclarative de son comportement.

Empaquetage Il faut ensuite empaqueter la réalisation de l'AE sous la forme d'un élément déployable suivant le méta-modèle de la figure 3.5. Ce paquet doit contenir les artefacts suivants :

1. une description décrivant son nom, sa version, son auteur et contenant une petite explication ;
2. tous les codes compilés (des AE primitifs, des codes de contrôleurs spécifiques et les réalisations des membranes) ;
3. toutes les architectures et les descriptions des points d'interaction ;
4. la configuration par défaut (des attributs) ;
5. une description et la configuration des exigences en terme de services de la plate-forme.

En général, lorsque le déploiement est pris en compte dans un modèle de composant, tous ces descripteurs sont des fichiers XML. Par exemple, en CCM, il y les fichiers : `cpf` pour la configuration des attributs, `csd` pour la description du paquet, `ccd` et `cad` respectivement pour les descriptions des composants et des composites (*assembly* pour CCM).

Une fois le paquet déployé sur un site hôte [9], automatiquement ou manuellement, son code binaire est chargé et les services de la plate-forme qu'il requiert sont démarrés et configurés pour l'AE. Parmi les services qui doivent être configurés, celui qui prend en charge l'instanciation de composant a un rôle important. Il s'agit, en général, de déclarer une nouvelle usine (*factory*)

9. Ici, on abstrait la localisation physique. Un hôte correspond à une plate-forme d'exécution qui peut, elle-même, être centralisée ou répartie.

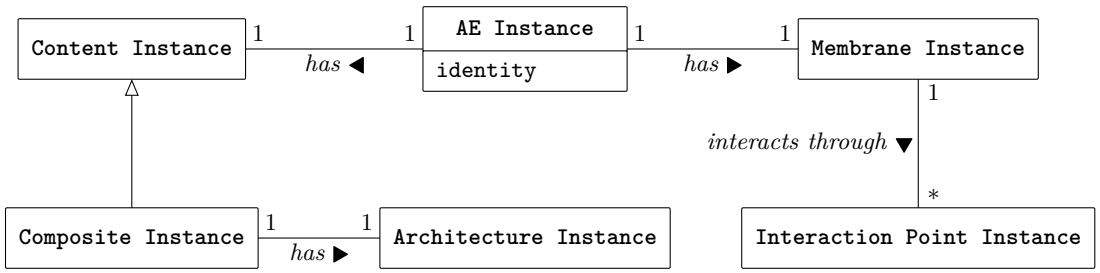


Figure 3.6 – Les instances d'élément architectural

réalisée pour un élément spécifique. Remarquons que dans la plupart des plates-formes, il est possible de changer la valeur des paramètres de configuration pour les adapter à une machine hôte spécifique.

Dans certains modèles, comme D&C [10] de l'OMG [P84], le paquet peut contenir plusieurs réalisations d'un même AE repoussant le choix de la réalisation effective au moment du déploiement sur une machine hôte.

Enfin, certains modèles de composant utilisent un *dépot de paquets* dans lequel sont chargés les composants qui deviennent alors accessibles à tous les sites qui connaissent le dépot. Cela rend le déploiement plus facile car il suffit alors de communiquer l'identifiant du composant au site destination qui peut alors obtenir le paquet depuis le dépot.

Instanciation Finalement, une instance représente un AE à l'exécution et contient son état. L'usine, évoquée lors du déploiement, est chargée de créer les instances qui deviennent alors accessibles par les autres instances. Remarquons que souvent l'instanciation est un processus récursif pour les AE composites, dans ce cas toutes les instances des sous-éléments doivent être crées puis reliées et enfin démarrées.

Le méta-modèle de la notion d'instance d'AE est représenté en figure 3.6. Les instances ont un identifiant pour que l'on puisse interagir avec une instance précise. Certaines plates-formes comme les serveurs d'EJB, enregistrent chaque instance créée dans un registre qui permet ensuite d'y accéder directement depuis le registre à partir de leur identité.

Remarquons que dans le modèle général présenté ici, toutes les formes d'instances sont représentées à l'exécution, la plate-forme peut ainsi aisément offrir une introspection de ces éléments architecturaux et ainsi permettre de la reconfiguration à chaud [11]. Parfois ces instances sont regroupées ou éliminées pour optimiser l'exécution, cela rend alors difficile d'offrir de l'introspection et de la reconfiguration.

3.3 Qu'est-ce qu'un composant ?

Les composants sont les éléments architecturaux offrant des services interfacés (*interfaced services*). Ces services interfacés sont des services requis ou fournis dont le contrat syntaxique est connu. Dans la classification en quatre niveaux des contrats de Beugnard et al [P19], un contrat syntaxique est défini comme un contrat qui permet la compilation et donc correspond à la notion d'interface *à la IDL* (un ensemble d'opérations).

10. C'est une généralisation de la précédente spécification du déploiement de CCM.
11. La reconfiguration à chaud fait l'objet du chapitre 5 page 71

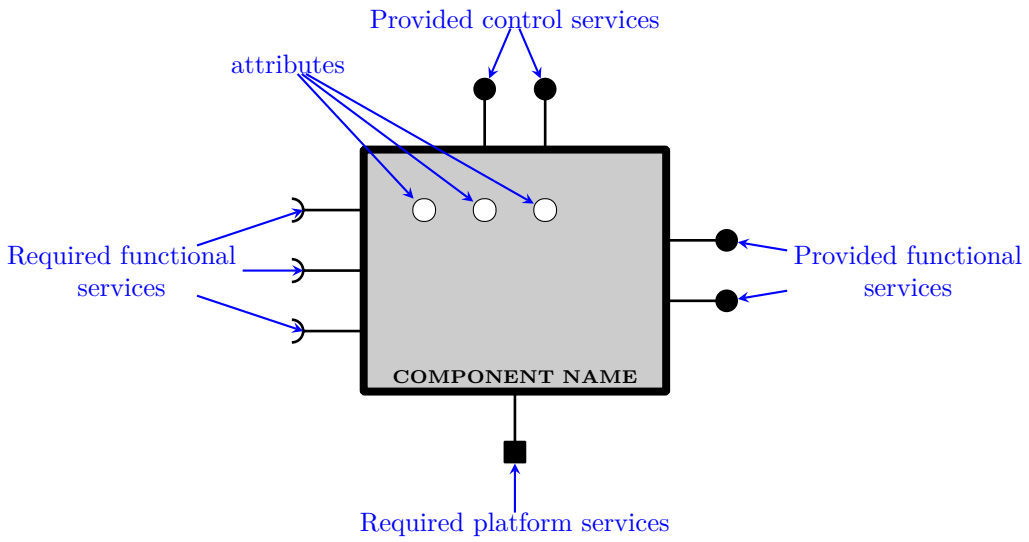


Figure 3.7 – Une représentation simple de la notion de composant

La spécification de composant Un composant étant un AE, pour le spécifier, il faut décrire ses attributs, son contenu et sa membrane. Ensuite, il faut raffiner la spécification de ses services pour définir leurs interfaces. On obtient ainsi un composant qui requiert et fournit des services fonctionnels [12], il fournit des services de contrôle et exhibe des contrats non fonctionnels. Un tel composant pourrait être représenté (graphiquement) comme sur le schéma de la figure 3.7. Traditionnellement, les services requis apparaissent sur le côté gauche, les fournis à droite, les services de contrôle sur le dessus et les contrats non fonctionnels [13] sur le dessous [14]. Le contenu du composant est grisé pour exprimer que les composants (primitifs) sont des boîtes noires et cachent leur contenu. Enfin, les attributs sont encapsulés et ne peuvent donc être manipulés que si un service de contrôle offre explicitement cette possibilité, soit de manière générique pour manipuler n'importe quel attribut par son nom (à la Fractal) ou par un service spécifique à chaque attribut (à la CCM).

Côté composant les services utilisent tous des *interfaces* pour contrat comme spécifié dans la figure 3.8 page suivante. Ces interfaces sont des ensembles d'éléments (opérations ou événements) typés. Cela permet ainsi de vérifier si un service fourni par un composant peut être utilisé par un autre composant (comme service requis et à travers un connecteur). Attention à ne pas confondre et mélanger les concepts d'`Interfaced Service Specification` et celui d'`Interface` car dans de nombreux modèles la spécification de service va bien au delà de la simple interface. Par exemple, en Fractal ou CCM, un service peut être obligatoire ou bien optionnel, simple ou multiple. Ces contrats font partie du contrat d'usage du service (voir figure 3.3 page 29).

12. Certains modèles de composant, comme AADL, n'ont pas de notion de requis et fourni mais un « service » peut avoir des exigences et en même temps fournir des opérations. Ces modèles où une interface mélange requis et fourni ne sont pas usuels, nous avons donc décidé de suivre la pratique la plus commune qui consiste à séparer ce qui est requis de ce qui est fourni.

13. Ils sont représentés par des carrés parce qu'ils peuvent être requis par le composant mais aussi il peut les fournir.

14. Ici, comme dans la plupart des modèles de composant récents, un composant peut avoir plusieurs contrats. Certains modèles plus anciens ou plus basiques restreignent le nombre de contrats possibles. Par exemple, un EJB n'a pas de contrat d'exigence, un unique contrat fonctionnel fourni et un unique contrat de contrôle (la maison *home*).

La partie haute vient du méta-modèle 3.3 page 29, les associations en pointillés sont indirectes

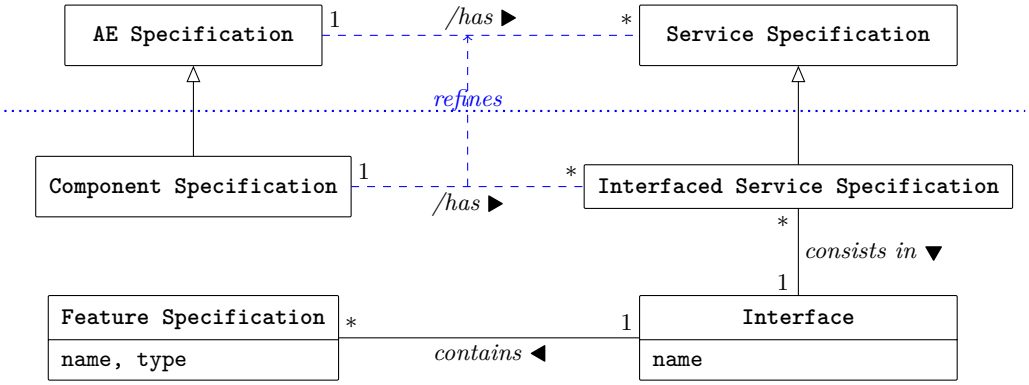


Figure 3.8 – La spécification de composant

Il convient de remarquer que la spécification d'un service peut soit être fournie par l'utilisateur (c'est un nouveau service utilisateur) soit fournie de manière standard dans le modèle de composant. Par exemple, en Fractal, une partie des services de contrôle sont prédéfinis comme le contrôleur de cycle de vie ou le contrôleur d'attribut.

Le langage utilisé pour définir les composants et les spécifications de service peut être un langage de programmation comme pour ArchJava mais aussi un langage plus abstrait (par rapport à l'exécution) et spécifique (dans son rôle) comme IDL3 pour CCM. Il est communément admis d'utiliser les termes de langage de définition de composant (*Component Definition Language* CDL) et d'interface (*Interface Definition Language* IDL). L'IDL3 de Corba est simultanément un CDL et un IDL. Fractal ADL est un CDL et utilise Java (les interfaces) comme IDL. Dans la figure 3.8, les entités `Interface` et `Feature Specification` sont définies en IDL alors que les autres entités sont définies en CDL. Cette séparation correspond aux deux unités de réutilisation que sont alors les composants et les interfaces. Ci-dessous, on peut trouver un petit exemple de spécification. Elle utilise des CDL et IDL simples et imaginaires juste dans le but d'illustrer le propos. Un composant `C1` et deux interfaces `I1` et `I2` sont spécifiés. Le composant `C1` contient deux attributs `a1` et `a2`, son contenu est spécifié dans le fichier `"tutu/titi/content.spec"`, ses deux services requis `s1` et `s2` sont tous deux d'interface `I1`, le service fourni `s3` d'interface `I2` et offre un service de contrôle du cycle de vie. Remarquez les exemples de contrats non fonctionnels (partie `nf contracts`) et d'usage (après le `with` dans les services).

```
component spec C1 {
  attributes
    int a1 ;
    string a2 ;
  content "tutu/titi/content.spec" ;
  nf contracts
    host kind = server  ;
    host cpu = powerful ;
    observables : memory, bandwidth ;
    persistency ;
  required services
    I1 s1 with security level = high ;
```

```
interface I1 {
  string method1(int a, int b) ;
  void method2() ;
}

interface I2 {
  string m() ;
}
```

La partie haute vient du méta-modèle 3.4 page 30, les associations en pointillés sont indirectes

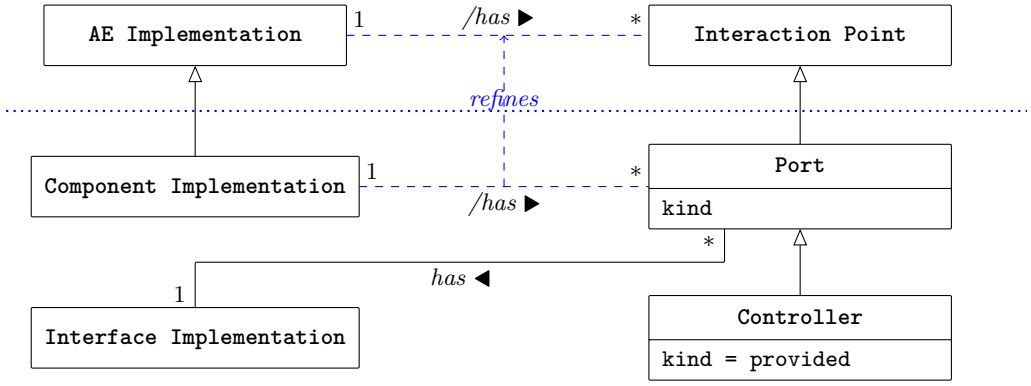


Figure 3.9 – L'implantation de composant

```
    I1 s2 with security level = low ;
  provided services
    I2 s3 with performance = high ;
  control services
    LifeCycleController lfc ;
}
```

L'implantation de composant Pour réaliser un composant, il faut implanter sa partie AE puis raffiner les points d'interaction comme formalisé par le méta-modèle de la figure 3.9. Pour un composant, ces points d'interaction sont des ports qui peuvent être requis ou fournis. Dans la plupart des modèles, les ports n'ont pas de réel comportement, ils contiennent seulement une référence sur un fournisseur externe (pour un service requis) ou sur un fournisseur interne (pour un service fourni). Mais, parfois, un port peut avoir un comportement plus spécifique par exemple, si l'on enrichit le modèle avec des interfaces optionnelles pour lequel il peut aussi gérer leur présence ou absence ([P59] par exemple étend Fractal avec une notion d'interface active qui fournit cette gestion du mode connecté / déconnecté [15]).

Les contrôleurs sont une catégorie spécifique de ports pour le contrôle du composant. Chaque contrôleur prend en charge un aspect de contrôle différent suivant le principe de la séparation des préoccupations. Dans certains modèle comme CCM, le type de ces contrôleurs est fixé par une API qui peut seulement être étendue en ajoutant des méthodes. Par contre, des modèles plus complets comme Fractal, offrent un *framework* complet pour la conception de nouveaux contrôleurs en définissant des interfaces par défaut qui peuvent être étendues. De plus, remarquons que certaines implantations du modèle Fractal comme Julia [P24] et AOKell [P102] peuvent être vue comme des *frameworks* de réalisation de contrôleurs. En effet, ils offrent des comportements par défaut pour le comportement des contrôleurs mais on peut choisir de les étendre.

15. Il est relativement étonnant qu'un composant, censé être autonome dans la vision traditionnelle, n'est pas toujours un comportement définis lorsqu'il n'est pas connecté...

La partie haute vient du méta-modèle 3.5 page 31, les associations en pointillés sont indirectes

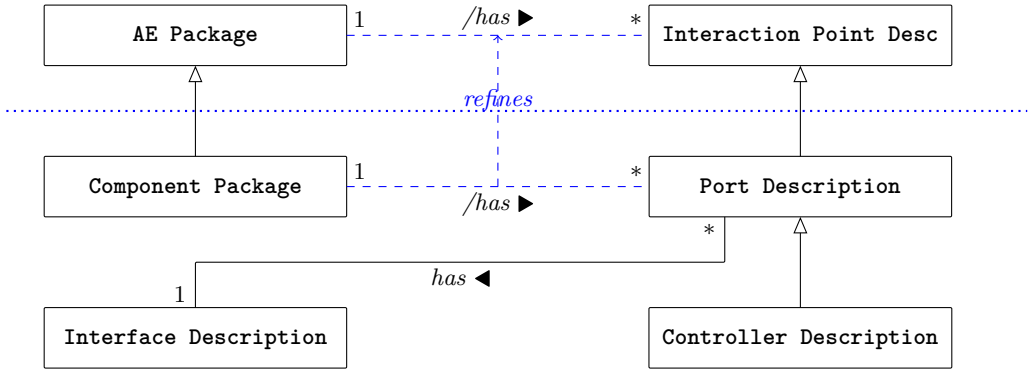


Figure 3.10 – Paquet de composant

La partie haute vient du méta-modèle 3.6 page 32, les associations en pointillés sont indirectes

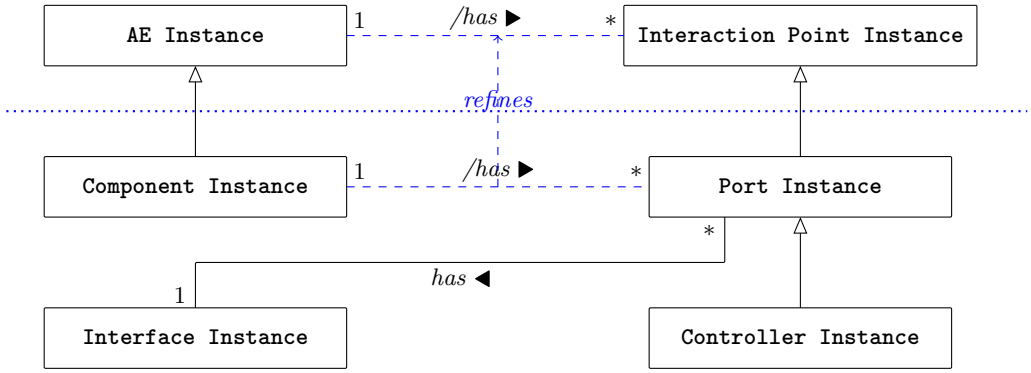


Figure 3.11 – L'instance de composant

Component package L'étape suivante consiste à empaqueter le composant pour pouvoir le déployer. Les descriptions des points d'interaction d'un composant sont raffinées en description de ports ou de sa spécialisation les contrôleurs. Un paquet suit le méta-modèle de la figure 3.10.

Component Instance Enfin, une instance de composant suit naturellement le méta-modèle de la figure 3.11. Notons que certains modèles de composant nomment conteneur la (instance de) membrane. Par exemple, en EJB et CCM, une unique entité, la maison (*home*), regroupe toutes les fonctions de contrôle.

3.4 Qu'est-ce qu'un connecteur ?

Comme cela a déjà été dit, un connecteur réifie l'interaction entre un ensemble de composants. Il peut être de bas-niveau et agir comme un simple fil de connexion (alors généralement implanté par un échange de références) ou de haut-niveau et réalisant des protocoles complexes (comme par exemple, du *publish / subscribe* ou un espace de *tuple*). Un grand nombre de connecteurs existent, il en existe une taxonomie que le lecteur intéressé peut consulter [P79]. Certains

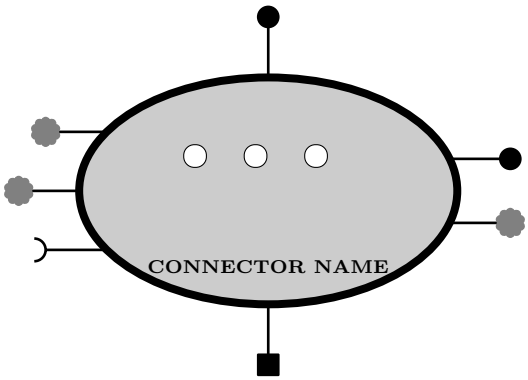


Figure 3.12 – Une représentation simple de connecteur

auteurs comme par exemple Lau dans [P68] vont un cran plus loin en déplaçant le contrôle du composant vers le connecteur qui devient alors l'appelant. Ici, nous ne franchissons pas ce pas et identifions seulement les connecteurs dans un objectif de séparation des préoccupations comme défendu dans [P13].

Étant un AE, un connecteur a quatre formes en fonction de la phase du développement à laquelle on le considère : une spécification de connecteur, une implantation de connecteur, un paquet de connecteur et enfin une instance de connecteur. De plus, il existe deux formes d'abstraction de communication :

1. Un composant de communication n'a que des interfaces (usuelles) définies. Un tel connecteur est en fait un composant mais est réifié comme connecteur pour séparer clairement le traitement métier de la gestion des interactions.
2. Un connecteur qui offre des *services malléables*. De tels connecteurs ne peuvent pas être spécifiés de la même façon que les composants. Deux exemples bien connus de tels connecteurs sont les connecteurs Corba et ceux de RMI. Ces connecteurs offrent des services dont la forme (l'interface) précise n'est pas connue avant qu'ils ne soient connectés à un composant. Dans le cas de Corba (et RMI), la réalisation de ces interfaces malléables sont le couple talon / squelette (*stub / skeleton*).

Certains modèles de composant ne supportent pas la notion de connecteur comme entité de plein droit comme Fractal par exemple. D'autres comme CCM et EJB offrent un nombre limité de connecteurs à travers les services offerts par leur intergiciel sous-jacent (ils offrent tous les deux par exemple de l'invocation à distance et de l'échange asynchrone de messages). Enfin, d'autres modèles plus sophistiqués comme SOFA supportent des connecteurs définis par l'utilisateur.

La représentation et les méta-modèles des connecteurs sont similaires à ceux des composants avec quelques légères modifications :

- Une `Service Specification` peut être spécialisée soit en une `Interfaced Service Specification` ou bien en une `Malleable Service Specification` (figure 3.13). Dans la figure 3.12, ces services malléables sont représentés par de petits nuages gris pour insister sur le fait que leur forme précise n'est pas connue quand ils ne sont pas connectés. Le contenu consiste principalement dans la gestion du protocole de coordination que le connecteur suit.
- Ensuite, un service malléable doit être réalisé sous la forme d'un `Role` (à la place d'un port). Ainsi, un connecteur peut avoir à la fois des ports et des rôles (voir 3.14 page suivante).

La partie haute vient du méta-modèle 3.3 page 29, les associations en pointillés sont indirectes

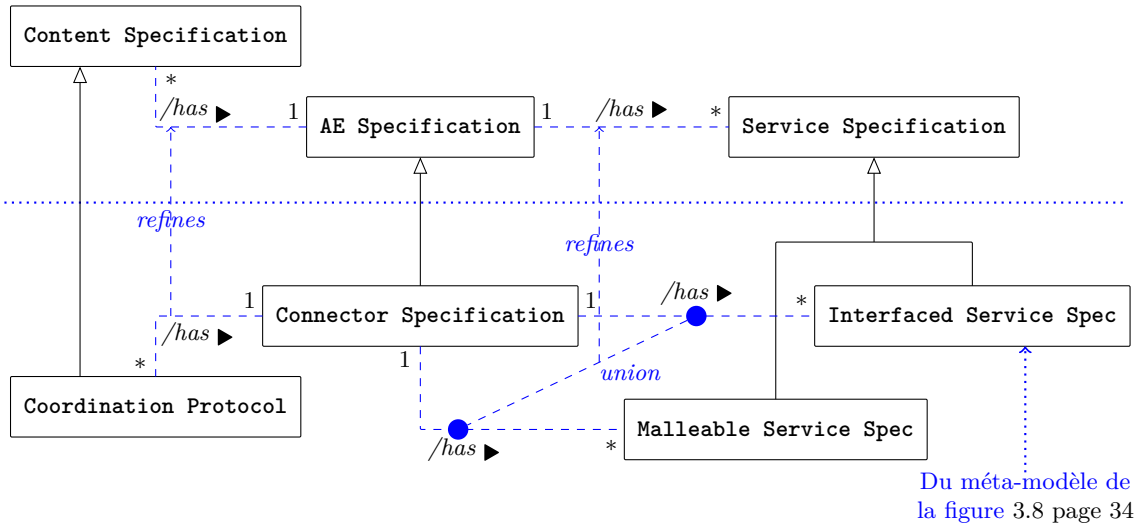


Figure 3.13 – La spécification de connecteur

La partie haute vient du méta-modèle 3.4 page 30, les associations en pointillés sont indirectes

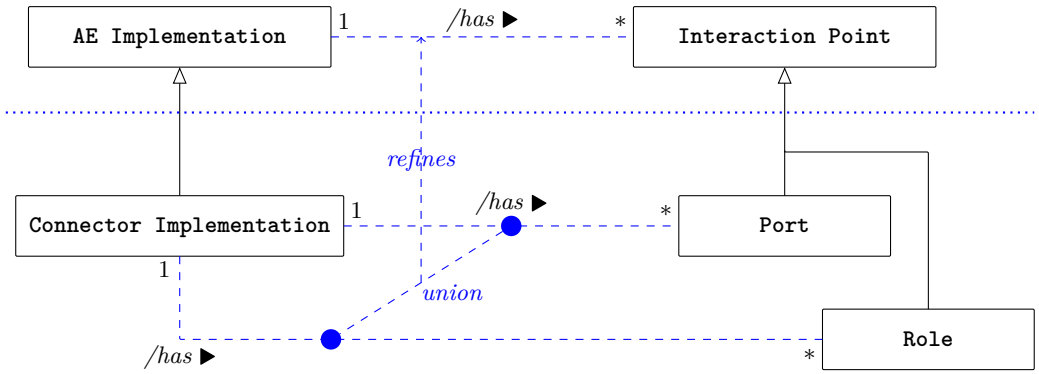


Figure 3.14 – L'implantation de connecteur

L'implantation d'un rôle malléable consiste général en un compilateur qui générera l'interface effective lorsqu'il sera connecté, [P77] contient de nombreux détails sur la spécification et la construction des connecteurs.

– Le déploiement est très proche de celui des composants, on y ajoute juste la description des rôles (voir 3.15 page ci-contre). Si la plate-forme d'exécution supporte la connexion à un rôle à l'exécution, le paquet de connecteur doit également contenir le générateur d'interface ou offrir un support de l'invocation dynamique. Les plates-formes Java EE modernes (qui exécutent des EJB) supportent la génération dynamique des talons et squelettes.

– Remarquons qu'un rôle devient un port lorsqu'il est connecté à un composant (généralement en utilisant un générateur). Mais de façon à supporter la déconnexion d'un rôle pour le connecter à un autre port, il faut garder le rôle derrière le port. Ainsi, dans la figure 3.16, une instance de connecteur peut avoir un port, un contrôleur ou un rôle. L'instance de rôle sera reliée à l'instance de port d'une instance de composant au sein d'une instance

La partie haute vient du méta-modèle 3.5 page 31, les associations en pointillés sont indirectes

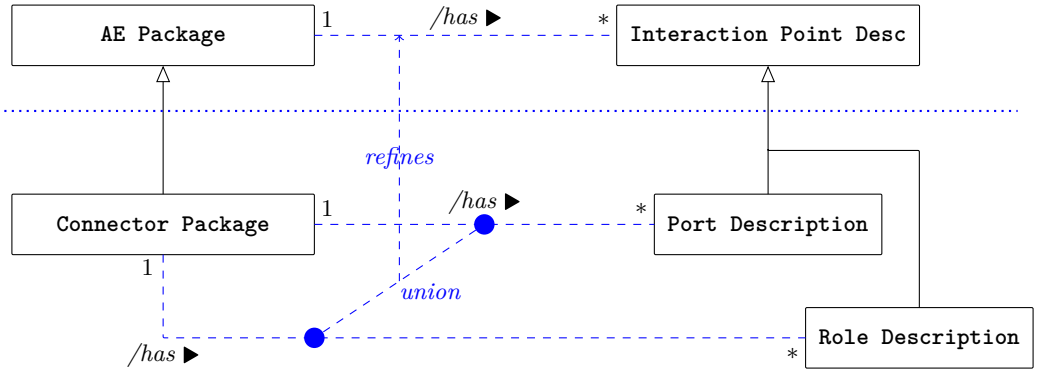


Figure 3.15 – Le paquet de connecteur

La partie haute vient du méta-modèle 3.6 page 32, les associations en pointillés sont indirectes

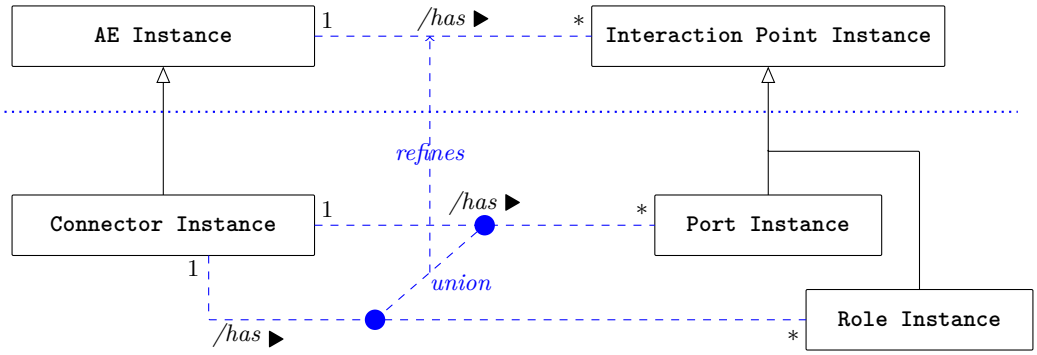


Figure 3.16 – L'instance de connecteur

d'architecture (présentée dans la section suivante). Et c'est grâce à ce lien (dynamique) que la plate-forme d'exécution connaît l'interface adoptée par le rôle.

Il convient de remarquer que la création de nouveaux connecteurs est une tâche plus ardue que celle de création de composant. Elle requiert généralement des développeurs de haut niveau. Par contre, il faut également souligner que la création de connecteurs spécifiques à un domaine offre une importante plus value car elle permet de grandement simplifier le développement des composants métiers. Il y a donc un compromis à faire.

3.5 Qu'est-ce qu'une architecture ?

La dernière notion qui doit être introduite est celle d'architecture qui permet de composer des composants et des connecteurs. Une instance d'architecture relie des instances de composants à travers des connecteurs. Une application (un composant composite) est construite par la plate-forme d'exécution en utilisant un *plan* appelé une architecture. Une architecture est donc une description d'un ensemble de *futures instances* (de composants et de connecteurs) ainsi que leurs attachements (certains modèles de composant utilisent la notion de *binding*). En

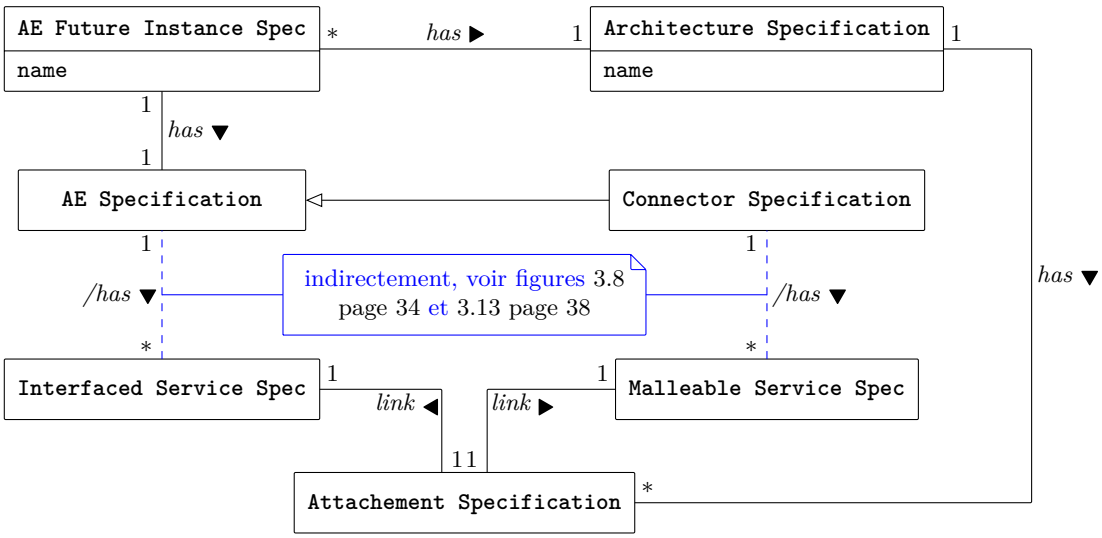


Figure 3.17 – La spécification d'architecture

général, une architecture est définie par un langage de description d'architecture (*Architecture Definition Language, ADL*) qu'il soit textuel ou graphique.

Les architectures vont à nouveau suivre le même découpage de leur description en fonction des phases du processus de développement. Elles commencent par être spécifiées (figure 3.17) en définissant un ensemble de sous-éléments architecturaux et en connectant leurs services. Cette spécification est ensuite raffinée pour définir une implantation. Le méta-modèle de la figure 3.18 page ci-contre formalise cette description. Remarquons que les futures instances d'AE sont logiquement nommées pour pouvoir être référencées par l'architecture (principalement pour réaliser les connexions). La fabrique qui va produire les instances exécutables des architectures à partir de cette description est en charge de produire toutes les instances des sous-éléments et de leur associer les identifiants de leurs instances.

Les services interfacés et les ports peuvent appartenir à des composants et à des connecteurs, ce qui explique l'utilisation des AE dans le méta-modèle. Dans la version finale des artefacts d'implantation, un port est toujours attaché à un rôle. Ainsi, un composant est attaché à des connecteurs alors qu'un connecteur peut être attaché soit à des connecteurs (pour ses ports) soit à des composants (pour ses rôles).

Ci-dessous, un exemple d'une telle architecture est donnée dans un ADL imaginaire. L'ADL peut inclure la spécification mais aussi l'implantation de l'architecture comme dans cet exemple. Ici, ces deux phases sont décrites sous forme textuelle. Ces descriptions sont assez similaires exceptées le fait qu'elles portent respectivement sur la spécification et l'implantation des AEs [16]. La représentation graphique proposée en figure 3.19 page 42 peut être utilisée pour les deux phases selon le choix fait sur les sous-éléments architecturaux. Remarquons que dans un outil de spécification, le concepteur pourrait mélanger des spécifications avec des implantations pendant son travail (il n'a réalisé qu'une partie des sous-éléments). Une vérification pourrait alors être

16. Il est possible que la réalisation soit bien plus différente de la spécification que l'exemple ne le sous-entend. Il pourrait, par exemple, y avoir plusieurs implantations de certains composants et certaines pourraient ne pas être connectées.

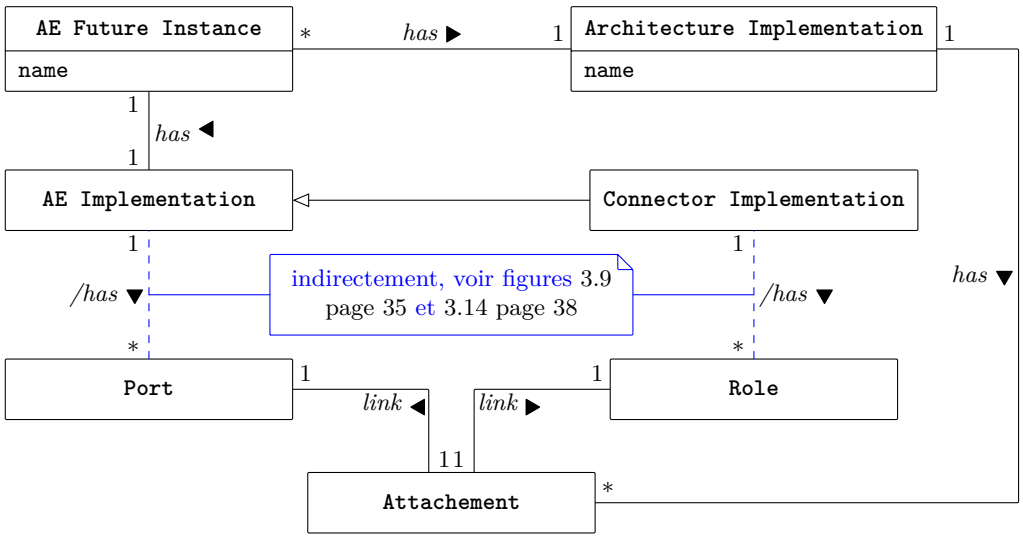


Figure 3.18 – L'implantation d'architecture

mise en place pour détecter si la réalisation est complètement terminée ou s'il reste des EA à implanter.

```
architecture spec Aspec {
  components spec
    C1spec comp1 ;
    C2spec comp2 ;
    C3spec comp3 ;
  connectors spec
    Co1spec con1 ;
    Co2spec con2 ;
  connections spec
    comp1.p1 -> con2.r ;
    comp1.p2 -> con1.r ;
    con2.r   -> comp3.p1 ;
    con2.r1  -> comp2.p2 ;
    con1.r2  -> comp2.p ;
}
```

```
architecture A implements Aspec {
  components
    C1 comp1 ;
    C2 comp2 ;
    C3 comp3 ;
  connectors
    Co1 con1 ;
    Co2 con2 ;
  connections
    comp1.p1 -> con2.r ;
    comp1.p2 -> con1.r ;
    con2.r   -> comp3.p1 ;
    con2.r1  -> comp2.p2 ;
    con1.r2  -> comp2.p ;
}
```

Cette spécification sous forme d'ADL est ensuite compilée en une représentation interne d'ADL (la *description d'architecture*) qui sera empaquetée dans le paquet de l'architecture (figure 3.20 page suivante) avec les paquets des sous-éléments architecturaux. Cette représentation interne sera utilisée par la fabrique pour instancier l'architecture (méta-modèle de la figure 3.21 page suivante).

3.6 Discussion

L'ensemble des méta-modèles présentés dans ce chapitre résulte d'une étude menée pour le compte d'Astrium et raffinée par les travaux menés chez Thalès Alenia Space. C'est une étude assez théorique mais qui permet de clarifier les notions de composant, connecteur et architecture. L'idée de représenter chaque avatar de la notion en fonction de la phase du développement est

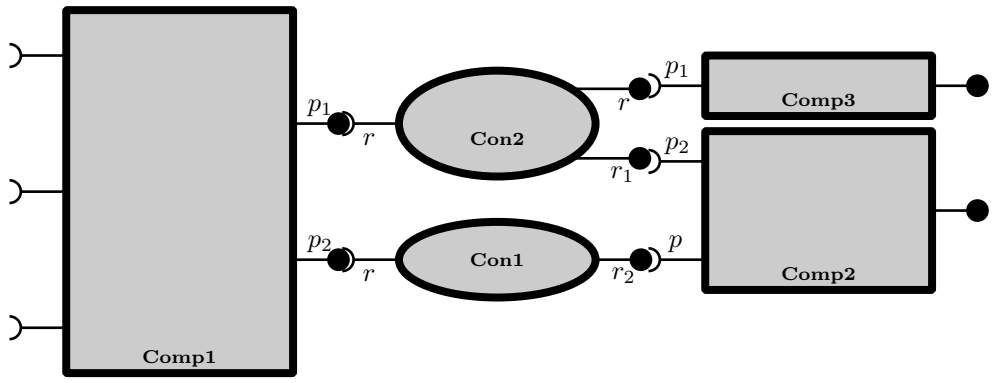


Figure 3.19 – Un exemple simple de représentation d'architecture

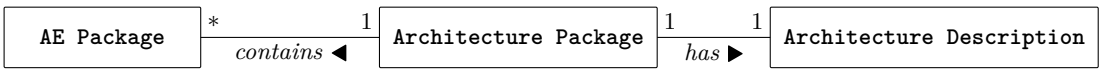


Figure 3.20 – Un paquet d'architecture

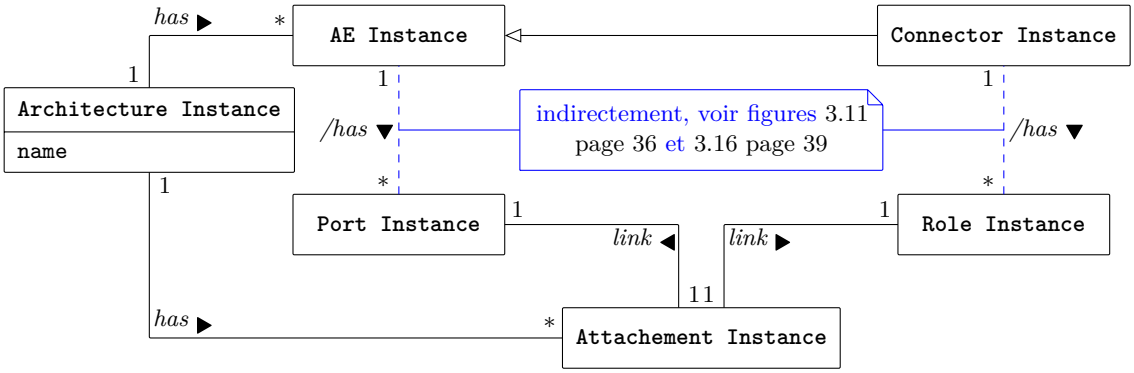


Figure 3.21 – Une instance d'architecture

née lors de mes recherches sur le déploiement et dans les nombreuses discussions avec Antoine Beugnard visant à bien comprendre la notion de composant.

Des expérimentations ont eu lieu pour valider cette proposition mais sont restées concentrées sur une partie seulement de ce qui est proposé ici. Plus précisément, les travaux menés avec Eveline Kaboré pendant qu'elle faisait un post-doctorat avec nous à Télécom Bretagne ont abordé la problématique de l'observation. Ces travaux publiés dans [34] proposent une méthode pour aider le développeur à intégrer de manière efficace un système d'observation à une application à base de composants Fractal. Pour cela, les composants affichent des contrats d'observation qui peuvent être requis (le composant veut observer quelque chose) ou fournis (le composant fournit une observation sur lui-même). Un processus de conception d'application sensible au contexte en trois phases a été proposé : (1) la spécification initiale identifie les composants et les observables, (2) la conception produit l'architecture à partir des choix faits par l'architecte et enfin (3) l'implantation effective de l'application incluant les choix technologiques. Un ensemble de méta-modèles sont définis pour supporter ce processus et un ensemble de transformations de modèle programmées en Kermeta l'outille. Ces transformations permettent d'assister la

réalisation de l'application entre chaque étape. Entre l'étape (1) et (2), le concepteur raffine son architecture en définissant les interfaces des services de ces composants et l'interface des services d'observation, la transformation génère alors l'architecture complète de l'application suivant un méta-modèle de Fractal (que nous avons développé). Entre les phases (2) et (3), une autre transformation de modèle génère le fichier Fractal ADL de l'architecture. Puis, à partir de la réalisation de la membrane choisie par le développeur, une dernière transformation génère le code d'implantation des interfaces d'observation en utilisant le *framework* d'observation COSMOS [P31]. Chacune des transformations contient des préconditions et postconditions qui assurent que chaque modèle (reçu ou produit) est conforme au méta-modèle attendu et respecte les choix de conception retenus dans la deuxième et la troisième étapes en particulier. Ce processus a été utilisé pour réaliser une version sensible au contexte d'un serveur HTTP existant écrit en Fractal : Comanche. On lui ajoute ainsi des capacités d'observation de l'utilisation du processeur et des mesures de l'utilisation d'un de ses services. Le processus est détaillé dans un rapport du projet [42].

Durant ces travaux nous avons validé l'importance de la séparation des préoccupations dans le cadre de processus de développement dirigés par les modèles. Cela a conforté l'idée qu'un processus doit reposer sur plusieurs méta-modèles spécifiques et être outillé de transformation pour aider à sa réalisation.

Cette approche a été également validée à nouveau dans le cadre d'un contrat avec Thalès en collaboration avec Christophe Guychard et Florent Diller. Ce projet a consisté à outiller l'ingénierie des interfaces (logicielles et matérielles) dans des systèmes de systèmes. Un méta-processus a été proposé reposant sur l'identification des préoccupations importantes pour le projet puis la construction de l'ordre de leur traitement en fonction de leurs dépendances. Ainsi, par exemple, une préoccupation transverse comme la sécurité ou la consommation en énergie intervient généralement en fin de processus alors qu'une préoccupation plus basique comme celle de la définition des interfaces externes intervient plutôt en début.

Ainsi, durant le projet, les composants du système vont avoir des interfaces dont les méta-modèles vont être adaptés aux différentes préoccupations abordées par le projet. L'approche choisie a été de proposer un méga-modèle [P38, P20], c'est-à-dire un modèle dont les éléments peuvent être des modèles. Ce méga-modèle est aussi vu comme un modèle de processus conforme à SPEM. Pour cela, chaque définition de produit (`Workproduct Definition`) se voit associée un méta-modèle lors de la définition du processus (avant la réalisation du projet) puis un modèle lors de la réalisation des tâches du projet. Un premier support outillé a été mis en place avec OpenFlexo pour, à partir de cette liste de préoccupations et des différents outils d'injection de leurs réponses, supporter l'exécution du processus construit. L'injection des solutions se base alors sur la fédération de modèles.

Ces travaux ont soulevé de nombreuses questions qui sont à la base de travaux faits par la suite, en cours ou prévus :

- Comment disposer d'un langage de description de processus qui permet de définir de tels processus utilisant de nombreux méta-modèles ainsi que des transformations de modèles ? Cette question a déclenché les travaux qui sont décrits dans le chapitre 6 page 111.
- Comment aller plus loin que la notion de méga-modèle et permettre de mélanger facilement des éléments de modélisation, des modèles, des méta-modèles de façon à pouvoir définir réellement efficacement et de manière générique des liens sémantiques entre ces entités ? C'est dans ce cadre que les travaux récents débutés sur la notion de fédération de modèles s'inscrivent. Ils sont récents et préliminaires et donc ne sont pas relatés en détail dans ce

document. Ils seront discutés rapidement dans la conclusion section 7.2.1 page 128 et lecteur interessé peut consulter [8].

– Pourrait-on utiliser les méta-modèles proposés dans ce chapitre, les combiner avec ce qui a été présenté sur le déploiement dans le chapitre 4 page ci-contre et ainsi obtenir un *vrai* bon modèle de composant réparti ? Cette question est encore en suspens mais j'espère avoir un jour le temps et les moyens de l'aborder.

Chapitre 4

Le déploiement

Dans le prolongement des travaux sur la gestion des dépendances et de ceux sur les modèles de composant décrits dans les chapitres précédents, j'ai également mené des travaux dans le cadre plus général du déploiement. De nombreux projets d'étudiants ont permis de construire différentes infrastructures et de tester leur utilisation. Deux thèses ont également eu lieu dans cette thématique, dans les deux cas les étudiants (Trinh Anh Tuan et Xu Zhang) ont abandonné avant le terme de leur thèse. Ces travaux sont donc restés un peu plus pragmatiques et ont été moins publiés que les précédents. Une proposition d'infrastructure pour le déploiement pair-à-pair est décrite dans [31].

Le but de ce chapitre est double : dresser un bilan sur des travaux pas complètement aboutis et leur donner une forme rédigée.

Les contributions de ces travaux sont les suivantes :

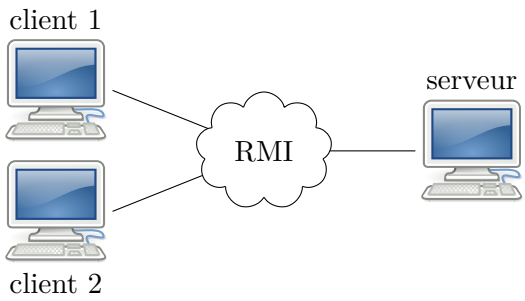
- Un processus de collecte de contraintes de déploiement qui suit une approche dirigée par les modèles.
- Un processus de réalisation du déploiement reposant sur la synthèse de plan de déploiement à partir des contraintes de déploiement affichés par le logiciel et de la description de l'infrastructure hôte.
- Un prototype de cette approche qui permet de déployer des applications distribuées composées de composants Fractal et OSGi. Ce prototype se concentre sur la synthèse des implantations et instances nécessaires pour satisfaire les contraintes de l'application.
- Une architecture et un modèle de méta-données pour le déploiement décentralisé de paquets.
- Une plate-forme pair-à-pair qui simule un réseau de déploiement suivant l'architecture proposée.

Le chapitre est naturellement organisé en deux sections, une sur le processus de déploiement des applications à base de composants et une sur le déploiement décentralisé de paquets.

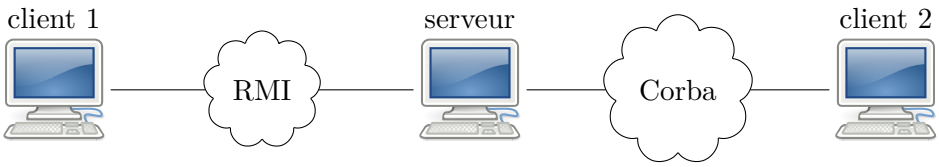
4.1 Vers un processus de déploiement pour les applications à base de composants

La première partie des travaux exposés dans ce chapitre concerne le processus nécessaire à la mise en place de déploiement automatisé pour des applications réparties à base de composants.

(a) L'architecture Fractal



(b) Première architecture concrète



(c) Seconde architecture concrète

Figure 4.1 – Une application Fractal simple et son déploiement

4.1.1 La motivation

Une application distribuée est construite comme une architecture d'assemblage d'un ensemble de composants. Le processus de déploiement classique des logiciels monopostes consiste à résoudre les dépendances de l'application puis à installer ces dépendances et à terminer par l'installation de l'application comme décrit dans le chapitre 2. Pour une application répartie, ce processus ne peut pas être réutilisé directement :

- La résolution des dépendances requiert la collecte des descriptions de toutes les machines cibles potentielles. Dans un contexte qui change régulièrement, cela reste un problème ouvert.
- Il faut choisir une architecture de déploiement pour l'application qui satisfait aussi bien les contraintes locales de chaque équipement que les contraintes plus globales du système.
- Les artefacts nécessaires à chaque composant déployé doit être amené sur la machine choisie pour héberger le composant en question.

L'approche usuelle [P15, P41, P60, P33] consiste à utiliser l'architecture comme guide pour le déploiement. Chaque (instance de) composant est déployé sur un hôte et les interactions entre composants réutilisent un intergiciel. Par exemple, une application Fractal simple est présentée figure 4.1 (a). Cette application est composée d'un composant serveur S et deux composants clients C_1 et C_2. Ces composants (clients) communiquent avec le serveur par l'intermédiaire de FractalRMI, une extension de Fractal qui utilise RMI pour les connexions entre composants distants.

Déployer une telle application sur une architecture concrète compatible telle celle de la figure 4.1 (b) est assez facile. Les composants C_1, C_2 et S sont installés respectivement sur les machines client 1, client 2 et serveur. En revanche, il n'est pas possible de simplement déployer cette application sur une architecture concrète un peu différente. Ainsi, pour l'architecture de la figure 4.1 (c), il y a plusieurs possibilités :

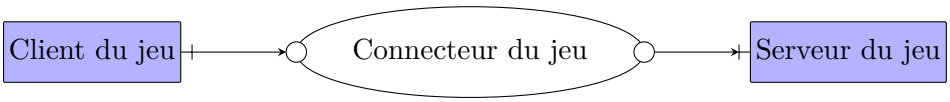


Figure 4.2 – Une architecture générique pour jeux massivement multi-joueurs

1. C_1 et C_2 peuvent s'installer sur la même machine et alors le déploiement ($C_1, C_2 \mapsto$ client $1, S \mapsto$ serveur) est possible ;
2. C_1 et C_2 doivent s'installer sur des machines différentes et aucune installation n'est possible ;
3. C_1 et C_2 doivent s'installer sur des machines différentes et on peut également utiliser Corba à la place de RMI alors le déploiement ($C_1 \mapsto$ client $1, C_2 \mapsto$ client $2, S \mapsto$ serveur) est possible.

Plus généralement, deux descriptions sont nécessaires ici. Le développeur de l'application doit décrire les contraintes *métier* de l'application qu'il a réalisée. De son côté le déployeur connaît bien l'environnement dans lequel il va déployer l'application. Il peut ainsi décrire comment il veut physiquement placer les composants tout en respectant les contraintes métier.

Nos travaux se sont déroulés dans le cadre des jeux massivement multi-joueurs (MMG), nous utiliserons des exemples de ce domaine pour illustrer les propos.

Un jeu massivement multi-joueurs est une application distribuée s'exécutant sur un réseau. Son architecture générique, présentée figure 4.2, consiste en un ensemble de clients qui se connectent sur un serveur de jeux pour rejoindre des sessions de jeu. Le développeur d'un tel jeu ne peut pas décrire le déploiement car il ne peut pas connaître le nombre de clients, leurs capacités, le nombre de serveurs et même les intergiciels de communication car ils dépendent des caractéristiques du réseau dans lequel l'application sera déployée [P75]. Il ne peut produire qu'une architecture abstraite dans laquelle les implantations et le nombre des instances de chacun des composants ne sont pas fixes et les communications distantes sont abstraites sous forme de connecteurs. Le déploiement d'une application décrite sous cette forme est plus flexible que celui basé sur une architecture d'instances. En effet, le choix des implantations, le calcul du nombre d'instances, leur placement et le choix des intergiciels de communication peut être repoussé à la phase de déploiement où l'architecture physique réelle sera connue.

Néanmoins, déduire la configuration concrète d'une architecture abstraite est difficile. Premièrement, la correspondance entre les éléments d'une description abstraite et les unités de déploiement n'est pas simple. Deuxièmement, il faut permettre l'ajout de contraintes de déploiement riches qui permettent au développeur d'expliciter les configurations possibles. Ainsi par exemple, le développeur devrait pouvoir exprimer des contraintes de la forme :

- Une instance du serveur de jeu est unique sur son hôte.
- Au moins deux instances de serveurs doivent exister : une, active sur laquelle les joueurs se connectent et une, de secours.
- Une implantation adaptée est déployée sur chacun des équipements d'un abonné.
- Chaque instance doit pouvoir se connecter à une instance de serveur active et une instance de secours.
- Un serveur de jeu ne peut servir simultanément qu'un nombre borné de clients fonction de ses capacités.

Ces contraintes forment un modèle abstrait des emplacements et de la distribution des instances sur ces emplacements. Ainsi, *équipement d'un abonné*, *serveur de jeu actif* et *serveur de jeu de secours* ne sont pas des emplacements physiques mais des nœuds virtuels de déploiement. La configuration concrète d'exécution de l'application doit bien sûr satisfaire ces contraintes.

4.1.2 La collecte des exigences de déploiement

Le cadre s'est construit en partie sur les résultats des travaux présentés dans le chapitre 3. Ainsi, l'architecture du logiciel consiste en un ensemble de composants reliés entre eux à travers des connecteurs. Le composant offre des ports, le connecteur des rôles et un port n'est relié qu'à des rôles. L'architecture, ses composants et ses connecteurs sont décrits par des modèles. Les composants et connecteurs ont des implantations. Une telle architecture sera décrite sous le nom d'*architecture abstraite*. La figure 4.2 page précédente décrivait donc une architecture abstraite.

Un ensemble d'activités sont nécessaires pour collecter toutes les contraintes de l'application qui peuvent intervenir lors de son déploiement. Pour cela, le producteur du logiciel doit raffiner son architecture abstraite pour en faire une *architecture de déploiement*. Ce modèle doit permettre de déterminer si un déploiement est possible et, si c'est le cas, permettre sa réalisation. Pour la première responsabilité, le modèle décrit en détail les contraintes que le site de déploiement de l'application doit respecter. Il contient deux formes de contraintes. Premièrement, des contraintes locales que chaque élément de l'architecture impose à ses futurs hôtes. Deuxièmement, des contraintes sur la variabilité de l'architecture globale (nombre d'instances ou topologie de connexion par exemple). Pour la seconde responsabilité, le modèle doit contenir des liens vers tous les artéfacts nécessaires au déploiement (fichiers d'implantation par exemple) ainsi que les paramètres importants à la construction de l'architecture (IPs et ports par exemple).

Dans ces travaux, nous avons quelques hypothèses :

1. Une instance de composant ne peut se déployer que sur un équipement. Un connecteur peut, lui, être réparti sur plusieurs équipements.
2. Les instances de composants peuvent être reliées soit directement soit à travers des *services de communication*. Dans le premier cas, les deux instances doivent être co-localisées (être dans le même environnement d'exécution).
3. Une architecture de déploiement supporte la variabilité du déploiement principalement au niveau des composants. Les composants peuvent avoir différentes implantations mais la topologie de connexion ne supporte que peu de variations [1]. La variabilité autorisée concerne une éventuelle répétition d'un sous-ensemble de l'architecture.
4. Les composants sont des boîtes noires pour le déploiement, le modèle est donc plat pour les composants. En revanche, les connecteurs peuvent avoir une structure hiérarchique complexe.

Ainsi, une architecture de déploiement n'est plus composée que de composants et de services de communication. C'est donc un modèle plat. Ces éléments présentent des exigences non-fonctionnelles qui contraignent le site déploiement. Les variations possibles sont les suivantes :

- le nombre d'instance des composants,
- la localisation de chacune des instances,

1. Probablement qu'une telle extension ne serait pas très complexe mais cela reste une intuition qu'il faudrait valider.

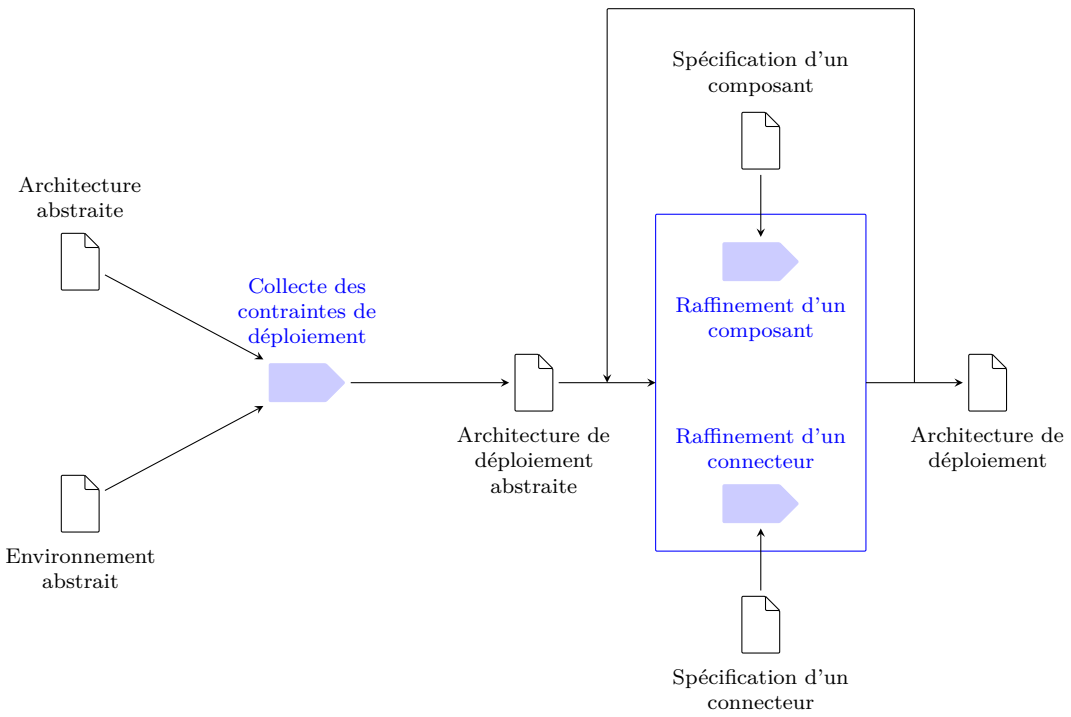


Figure 4.3 – Production de l'architecture de déploiement

- l'implantation de chacune des instances,
- la contingence de l'installation de chacune des instances [2],
- le nombre de ports des composants,
- la configuration des services de composants,

Pour la partie localisation, la notion d'*environnement de déploiement abstrait* permet de décrire une notion abstraite de *nœud*.

La figure 4.3 décrit ce processus de construction de l'architecture de déploiement. Pour indiquer que le bloc des activités de raffinement est incrémental et itératif, nous avons ajouter une flèche qui reboucle son entrée avec sa sortie. Ce processus suit une approche dirigée par les modèles. Les artéfacts sont donc tous des modèles et les activités sont des transformations (lorsqu'il est nécessaire de changer de méta-modèle par exemple) et raffinements de modèle. Les deux sous-activités de raffinement visent à rendre le modèle plat (sans hiérarchie) et sans connecteurs. Notons également que la sémantique fonctionnelle précise des composants et connecteurs ne concerne pas le déploiement. Seuls les parties de cette sémantique qui contraignent le déploiement sont intégrées dans l'architecture de déploiement. Par exemple, les propriétés d'un algorithme peuvent contraindre le nombre de clients qu'un serveur peut gérer. L'algorithme ne concerne pas le déploiement mais la contrainte de taille si. Les formes de communication peuvent également contraindre le déploiement si, sur un nœud, le service de communication nécessaire n'existe pas. Cette contrainte doit donc apparaître dans l'architecture de déploiement.

2. Ici, on veut dire que la création d'une nouvelle instance peut être imposée, possible ou interdite. Ainsi lors du déploiement sur le site client, la pré-existence d'une instance peut être imposée (interdite). D'autre part, lors de l'installation, la création d'une instance doit être faite (imposée) ou choisie (possible).

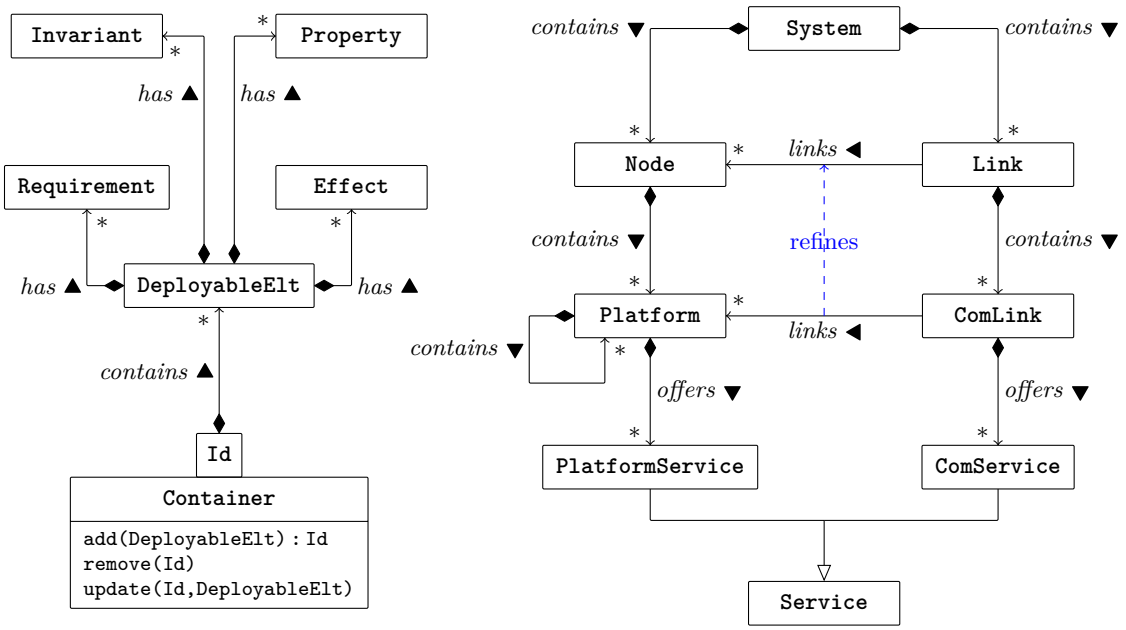


Figure 4.4 – Les méta-modèles de déploiement et d'environnement de déploiement

Pour les composants, il y a deux situations possibles. Soit le composant est vraiment en boîte noire et dans ce cas, le raffinement ne fait rien à part inclure les propriétés et implantation du composant dans l'architecture de déploiement. Soit un des sous-composants doit être connu du système de déploiement, par exemple, s'il ne doit pas être instancié (mais doit déjà être présent sur le nœud cible). Dans ce cas, il faut aplatir la hiérarchie pour le faire apparaître [3]. Notons pour finir sur les composants que l'on ne connaît pas encore les caractéristiques précises des hôtes des différents composants. Il n'est donc pas possible de choisir une implantation. Ainsi, les composants peuvent, à ce stade, être fournis avec plusieurs implantations.

Pour les connecteurs, le travail est un peu plus important car il faut les faire disparaître. Il va donc falloir intégrer l'architecture interne du connecteur s'il en a une et choisir les services de communication que le site de déploiement devra fournir [P78].

Enfin, remarquons que dans une architecture de déploiement il ne reste plus que des composants qui vérifient :

- Si deux composants sont connectés directement (un port de l'un est relié à un port de l'autre) alors la communication est locale et ils sont donc forcément co-localisés.
- Si deux composants sont reliés par un service de communication mais qu'ils sont instanciés dans le même conteneur alors on peut basculer à une connexion locale. Notons que pour que le déploiement soit possible, aucune contrainte ne doit s'opposer à cette co-localisation.

4.1.3 Le déploiement

De son côté le client doit, à partir de l'architecture de déploiement, calculer ou concevoir l'ensemble des *configurations concrètes* possibles. Pour cela, il s'agit de trouver ou choisir des

3. Des exemples de telles architectures peuvent se rencontrer par exemple dans le modèle Fractal ou une instance de composant peut être partagée par plusieurs composants englobants.

hôtes pour chaque élément de l'architecture ainsi que l'implantation adaptée à ces hôtes. Une configuration est alors une architecture ne contenant plus que des instances de composants avec leur localisation effective. Toutes les variations ont été résolues, c'est-à-dire, un choix a été fait. Pour cela, le client utilise un modèle d'*environnement de déploiement* qui décrit son infrastructure. Dans le cadre de nos travaux, nous avons utilisé le méta-modèle de la partie droite de la figure 4.4 page ci-contre pour décrire les environnements de déploiement. Il repose sur un patron pour exprimer la contenance qui est décrit en partie gauche. Un élément déployable possède des exigences et des effets pour son déploiement (sur le modèle de la notion de dépendance du chapitre 2), des propriétés et des invariants. Un invariant doit être vrai tout au long de la vie de l'élément. Ces éléments peuvent être déployés dans un conteneur. Dans ce cadre, un système est un conteneur pour des nœuds et des liens qui représentent respectivement des machines et des liens réseaux entre elles. Au sein des nœuds, on peut déployer des plates-formes qui elles-mêmes peuvent contenir des plates-formes. Par exemple, on peut avoir une machine virtuelle qui contient un serveur Tomcat. Chaque plate-forme peut fournir un ensemble de services que les composants pourront requérir et utiliser. Par exemple, Tomcat fournit le service `websocket`. Suivant un modèle similaire, il est possible de déployer sur les liens, des liens de communication qui offrent des services. Par exemple, un intergiciel de gestion d'événements pourrait être mis en place sur un lien et offrir un service d'abonnement suivant des types d'événements. Notons, que pour relier deux plates-formes par un lien de communication, il faut qu'elles aient un lien réseau entre elles.

Comme vu dans le chapitre sur la résolution des dépendances (chapitre 2), chaque conteneur doit maintenir suffisamment d'informations pour permettre de vérifier les opérations de déploiement. En effet, il faut vérifier que les exigences sont satisfaites mais également que l'opération ne va pas invalider l'un des invariants du conteneur. Le conteneur doit, entre autre, maintenir six ensembles : (1) les composants installés, (2) les services locaux disponibles, (3) les composants interdits, (4) les services interdits, (5) les services qu'un des composants du conteneur ou lui-même fournit aux autres conteneurs ou à leurs composants (services distants fournis) et (6) les services des autres conteneurs ou de leurs composants qu'il peut utiliser (services accessibles à distance). Enfin, lorsqu'un composant requiert un service à travers un service de communication, il peut imposer des contraintes. Il est possible de contraindre la localisation du fournisseur de service à être : (1) local, (2) distant n'importe où, (3) n'importe où ou (4) sur certains nœuds (par leur identifiant ou des propriétés ou capacités qu'ils doivent avoir).

Une fois l'ensemble des configurations possibles construit, l'*architecture d'instances* que l'on va déployer est choisie dans cet ensemble. Pour cela, on peut les comparer en utilisant les propriétés des différentes configurations possibles sur la base des caractéristiques exactes de chacun des éléments du site de déploiement. Il est alors possible de produire le *plan de déploiement* qui va orchestrer l'ensemble des opérations de déploiement. Ce plan est enfin exécuté pour rendre exécutable l'application. Parfois, l'ordre de réalisation des sous-opérations de déploiement est important. Si cette contrainte d'ordonnancement provient d'une exigence, elle est implicitement transmise dans le modèle de déploiement. En effet, chaque exigence devra être installée avant celui qui l'exige. Par contre, si ce n'est pas le cas, il faut introduire une information le permettant. Par exemple, un composant qui utilise des éléments installés par un autre composant. Nous

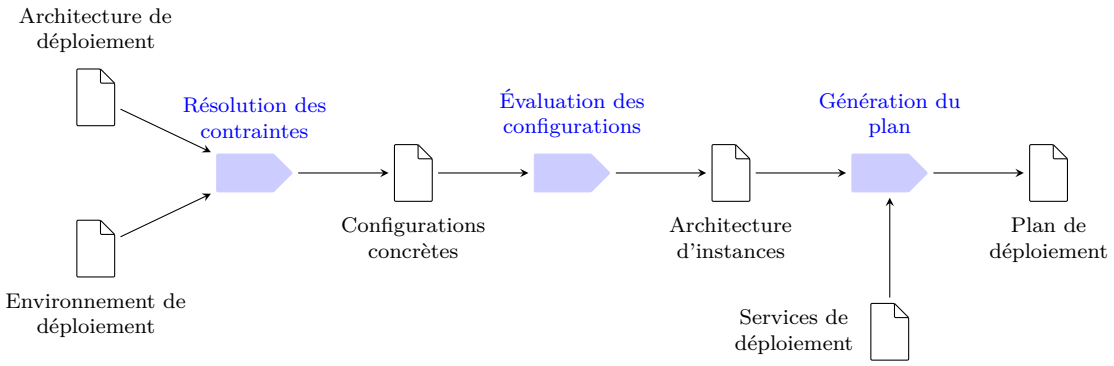


Figure 4.5 – Production du plan de déploiement

n'avons pas eu le temps d'explorer des solutions à ce problème [4] et notre prototype se contente de l'ordre des dépendances et de l'ordre de définition dans le descripteur d'architecture de déploiement.

Ce processus de déploiement côté client est présenté figure 4.5.

Remarquons que le gestionnaire de déploiement est responsable de la correction des étapes de déploiement. Il doit donc intégrer un mécanisme de transaction pour pouvoir annuler une opération qui n'a pas pu se dérouler entièrement correctement. Il est aussi en charge de la vérification du maintien des invariants de chaque composant. Pour cela, une structure hiérarchique des différents gestionnaires locaux de déploiement permet de résoudre localement les invariants. Si les invariants reposent sur des informations de composants distants, la collectivité des gestionnaires de déploiement doit se coordonner.

Enfin, les composants doivent disposer des bons attributs de configuration et de méta-données de dépendance correcte pour que les opérations de déploiement puissent se dérouler correctement. Nous n'avons pas du tout traité des cas où les données étaient insuffisantes ou les dépendances n'étaient pas correctes. Le prototype provoque des erreurs dans ce cas et le mécanisme de transaction n'a pas été complètement opérationnalisé.

4.1.4 Un exemple

Pour illustrer la première partie du processus, nous allons utiliser un exemple. Il s'agit d'un jeu MMG librement inspiré des expérimentations que nous avons menées. Le but est de concrétiser un peu plus le processus.

Le processus commence par l'architecture abstraite [5] décrite dans le haut de la figure 4.6 page ci-contre. Le serveur du jeu se compose de deux parties indépendantes : un composant d'authentification et un composant de gestion des sessions de jeu (les parties). La connection du composant de jeu côté joueur à ces composants serveurs se fait au travers de connecteurs spécialisés : un pour la connexion et l'autre pour l'accès à la session.

4. Peut être qu'une approche réifiant également d'autres éléments que les services requis pourrait résoudre ce problème. Par exemple, en OSGi, un composant fournit du code que d'autres composants co-localisés peuvent utiliser et requérir. On pourrait imaginer aussi pouvoir requérir une ressource comme une image ou un fichier.

5. Pour simplifier les schémas et le discours les ports et rôles ne sont pas typés ici. Dans notre prototype, ils sont bien sûr typés.

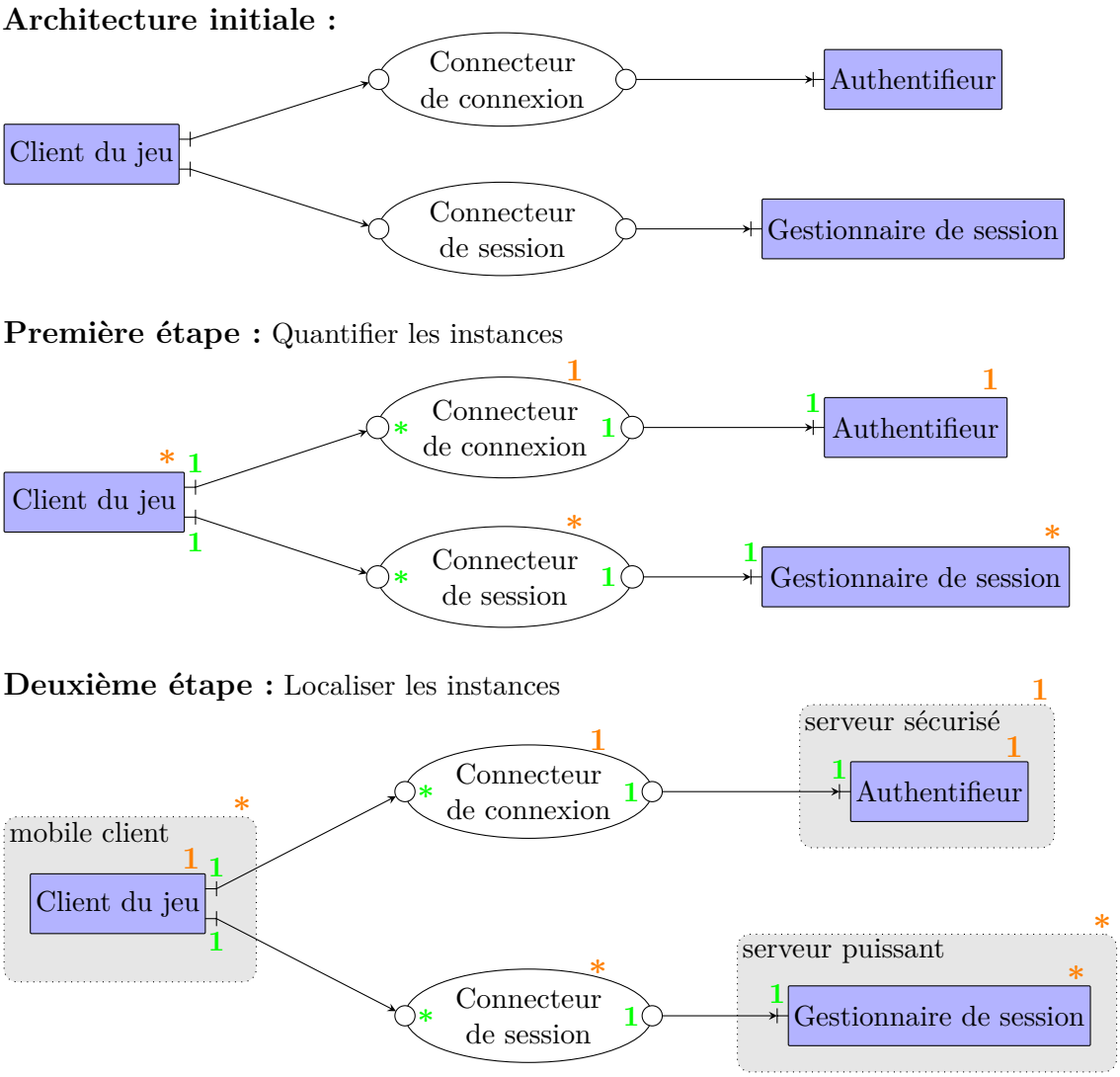


Figure 4.6 – Exemple de suivi du processus de construction de l'architecture de déploiement

La suite de processus consiste à commencer à construire l'architecture abstraite de déploiement. Pour cela, dans l'exemple, je procède en deux fois mais sur une application plus conséquente, il faudrait sans doute itérer un grand nombre de fois.

La première étape ajoute les contraintes sur les nombres d'instances. Ainsi, chaque composant, chacun de ses ports, chaque connecteur et chacun de ses rôles se voit attribuer une cardinalité. Le langage pour décrire ces cardinalités est classique. Dans l'exemple, on contraint le déploiement de l'application en autorisant un nombre quelconque de clients et de gestionnaires de sessions mais un unique composant en charge de l'authentification. De même, le connecteur de connexion sera unique alors qu'il pourra y avoir autant de connecteurs de session que nécessaire. Notons que vu les contraintes sur les nombres de ports et rôles, il y aura forcément autant de connecteurs de session qu'il y a de gestionnaires de session.

La deuxième étape étend l'architecture de déploiement abstraite avec les informations de localisation. Ainsi, il s'agit d'identifier des localisations (pour l'instant abstraite) et y affecter les

composants. Ses localisations vont avoir des exigences (par exemple de capacité), des cardinalités et des contraintes de co-localisation. Par exemple, deux localisations abstraites ne peuvent pas être sur la même machine physique pour améliorer la tolérance aux pannes de l'application. Dans la figure, nous n'avons pas indiqué de telles contraintes. Par contre, nous identifions trois localisations abstraites : le mobile du client, un serveur *sécurisé* et un serveur *puissant*. Des exigences de caractéristique pourraient également être ajoutées ainsi que des descriptions (instructions de choix) à ses localisations. Notons que les cardinalités des composants peuvent changer puisqu'elles sont maintenant relatives à une instance de localisation [6]. Ainsi, il peut y avoir autant de mobiles clients que l'on veut sur lequel n'est déployé qu'une unique instance du composant de jeu. Par contre, le nombre d'instances du gestionnaires de jeu sur chaque serveur puissant n'est pas contraint. Plus généralement, tout composant qui avait une cardinalité de n sans localisation, placé dans une localisation de cardinalité p, prend la cardinalité n' tel que $n' \times p = n$.

Une fois la collecte des contraintes de déploiement réalisée, il faut raffiner les composants et les connecteurs. Pour notre exemple, nous allons réaliser deux itérations. Elles sont nommées troisième et quatrième étapes dans la figure 4.7.

Dans la troisième étape, nous raffinons les trois composants et les deux connecteurs :

- Le composant d'authentification est choisi pour être implanté sous forme de *servlet* et utilisera la technologie SSL. Le connecteur qui permet au composant client de se connecter utilisera le service de communication HTTPS.
- Le connecteur de session est un élément complexe qui se voit remplacer par une architecture utilisant trois connecteurs et quatre composants. Le principe retenu est de séparer le flux (montant) d'événements venant des clients et contenant leurs actions de jeu (déplacement par exemple), du flux vidéo (descendant) produit par le serveur de session. Ainsi, les utilisateurs font des actions qui sont interprétées par le serveur qui calcule des nouvelles scènes de jeu qu'il encode en MP4 avant les rediffuser à tous les joueurs de la session [7].
- Les composants clients du jeu et gestionnaires de session sont raffinés pour réutiliser les composants issus du raffinement du connecteur de session.

Le schéma contient également une nouvelle notion, celle d'instance requise. Il s'agit des composants, à fond vert, d'encodage et de décodage vidéo. Ici, lors de son déploiement, le logiciel ne va pas instancier ces composants mais requiert qu'ils existent déjà de façon à se connecter à leurs instances existantes.

Enfin, la quatrième étape fait disparaître tous les connecteurs en les remplaçant par des services de communication. En l'occurrence, la diffusion d'événements utilise un intergiciel orienté message (RabbitMQ par exemple) alors que la diffusion vidéo reposera sur une diffusion RTP.

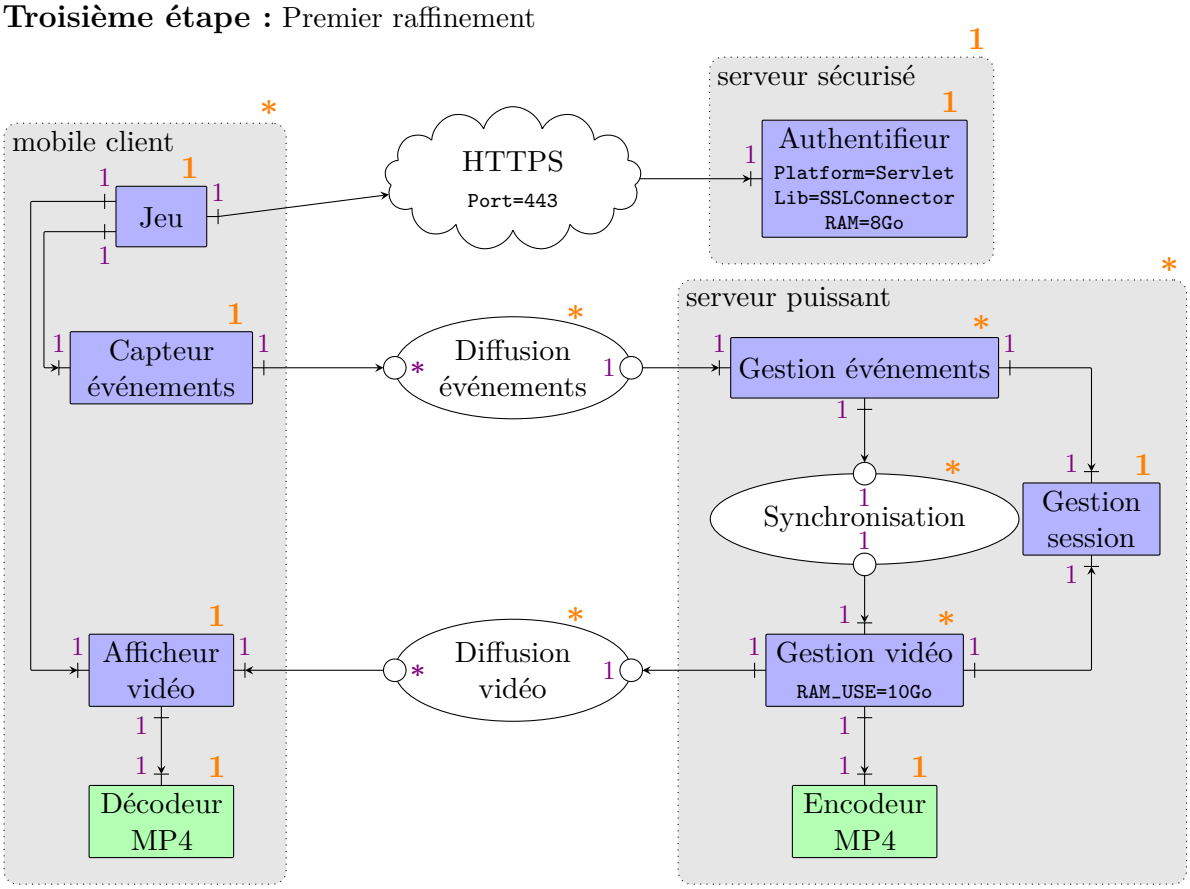
4.1.5 Le prototype

Nous avons réalisé un prototype simplifié qui aide à la partie réalisation du déploiement. Il s'agit d'une application distribuée capable de déployer des applications complexes composées

6. Au sein du prototype, nous supportons un niveau de contrainte de plus. Il est possible d'indiquer que l'instance du composant est dans (1) toutes les instances de la localisation, (2) au moins n instances de la localisation ou (3) toutes les instances de la localisation qui peuvent / veulent.

7. Nous n'avons pas travaillé sur les architectures de jeu mais nous en avons examiné quelques unes pour en faire des cas d'étude.

Troisième étape : Premier raffinement



Quatrième étape : Deuxième raffinement

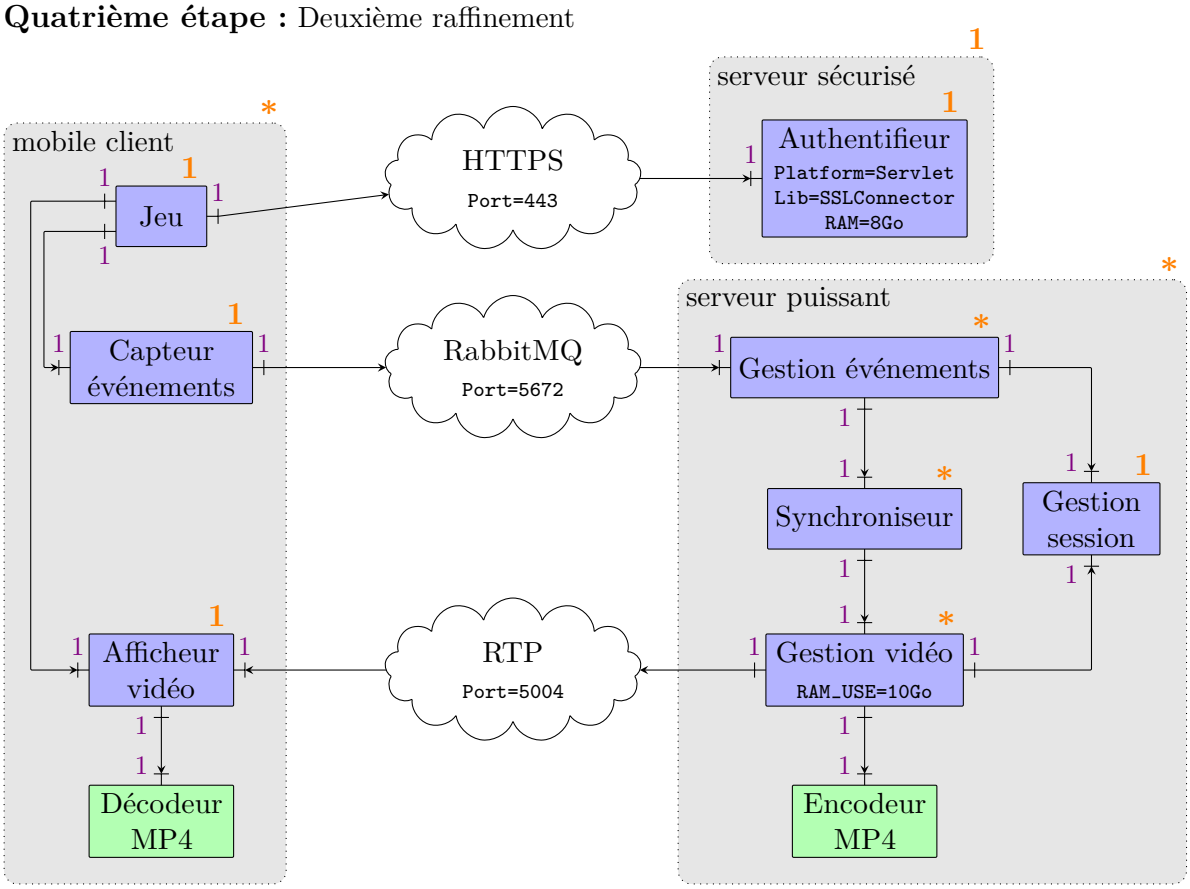


Figure 4.7 – Exemple de suivi du processus de construction de l'architecture de déploiement

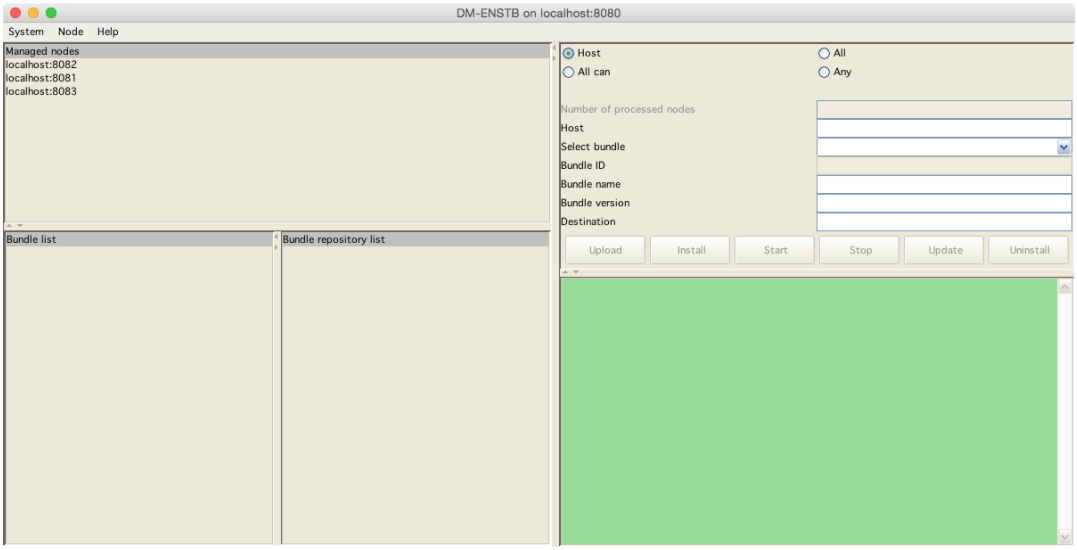


Figure 4.8 – L'interface du déployeur OSGi

de composants Fractal et de composants OSGi. Ce prototype suit une architecture maître-esclave pour la réalisation du déploiement. Un déployeur reçoit des commandes de déploiement depuis un utilisateur. Il supervise alors ces opérations de déploiement. Pour cela, il va utiliser l'ensemble des déployeurs d'autres nœuds qu'il connaît. Il diffuse à chaque nœud la description de l'application à déployer et les nœuds peuvent candidater au déploiement de tout ou partie de l'application. Le nœud chargé du déploiement choisit alors une architecture d'instance concrète qui satisfait les contraintes de déploiement de l'application. Une fois cette architecture d'instance calculée, le déployeur maître va envoyer des ordres de déploiement d'instances à chacun des nœuds sur lesquels ces instances doivent être déployées.

La construction de ce prototype s'est fait par étape. Nous avons tout d'abord expérimenté sur le déploiement réparti d'application OSGi. Dans ce cadre, nous supportons deux formes d'architectures : un fichier de configuration des nœuds OSGi définit l'ensemble des nœuds connus ou un protocole de découverte dynamique utilisant JXTA. La description des *bundles* est étendue pour inclure les exigences de déploiement. Cette application est elle-même livrée sous la forme de trois *bundles* : `DeploymentService` pour réaliser les opérations de déploiement, `DeploymentManager` pour gérer des déploiements répartis et `DeploymentGUI` qui offre une vue graphique globale du réseau de nœuds OSGi. L'ensemble des nœuds échangent des messages par XMLRpc. L'interface graphique est présentée en figure 4.8. Elle permet de visualiser la liste des *bundles* installés sur chacun des nœuds mais aussi d'installer des applications composées d'un ensemble de *bundles*. Le principe est qu'un *bundle* sert de point d'entrée et a en dépendance tous les *bundles* de l'application, c'est sa description qui contient les contraintes de déploiement. Cette application utilise le solveur de contraintes Caml présenté dans le chapitre 2 à travers un connecteur JNI [8].

Un déployeur de composant Fractal a ensuite été réalisé sur le même principe en se concentrant sur son architecture. Il a été lui construit sous forme de composant Fractal. On se contente

8. *Java Native Interface*, l'interface qui permet d'appeler des fonctions C depuis la JVM est utilisé pour déclencher les fonctions Caml du solveur de contraintes.

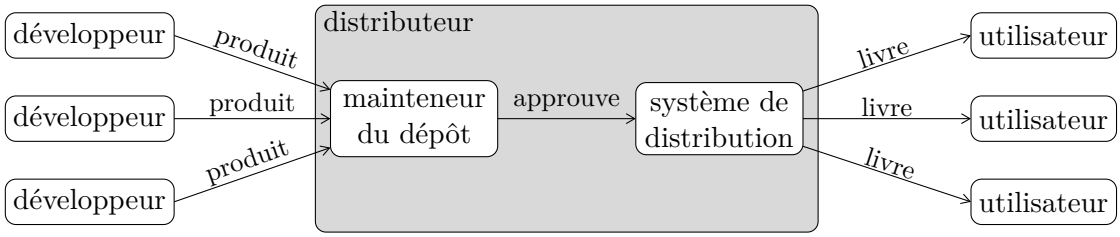


Figure 4.9 – Le processus typique de distribution de logiciels

d'une interface de type ligne de commande. Enfin, à partir de ces deux efforts, nous avons construit un prototype de déployeur Fractal / OSGi.

4.2 Le déploiement réparti pair à pair

En novembre 2014, GitHub annonce avoir 7,5 millions de comptes utilisateurs pour plus de 16,9 millions de dépôts [9]. BitBucket annonce en 2013 avoir dépassé le million de comptes [10]. Et Sourceforge héberge plus de 325 000 projets pour 3,5 millions de comptes utilisateurs [11]. Même si ces chiffres ne donnent pas une idée précise du nombre de développeurs de logiciel, on peut trouver les chiffres de 11 millions de développeurs professionnels et de 7,5 millions de développeurs amateurs [12]. Si l'on se concentre sur une petite partie de la production logicielle en regardant uniquement le nombre d'applications présentes sur les principaux marchés en ligne, on arrive à plus de 3 millions [13]. Et la croissance du nombre de ces applications continue sur rythme effréné d'environ 300 000 applications par an rien que sur l'AppStore d'Apple [14]. De l'autre côté, les clients téléchargent toujours plus de logiciels, Apple revendique 10 milliards de téléchargements entre juin et octobre 2014 et 100 milliards au total depuis 2008 [15]. On constate même que lors des mises à jour de système une part de plus en plus importante du traffic internet mondial est consacrée au téléchargement d'applications, pour iOS7 fin 2013, on était à 20 % du traffic [16].

Tous ces chiffres récents confirment une tendance à la base des travaux que j'ai mené dans le domaine entre 2009 et 2012, le logiciel est produit massivement, de plus en plus rapidement, par un nombre sans cesse plus important de développeurs. Dans ce cadre, le projet `dpan` (pour *Distributed Package Management Network*) a visé à proposer les bases de solutions de déploiement décentralisée et large échelle.

4.2.1 Le déploiement actuellement

La figure 4.9 présente le processus actuel de distribution de logiciels sous la forme d'un flot partant des *développeurs* vers les *utilisateurs* à travers les *distributeurs*.

9. https://github.com/about/press
10. https://blog.bitbucket.org/2013/06/04/atlassian-bitbucket-passes-one-million-users/
11. http://sourceforge.net/blog/sourceforge-myths/
12. http://www.infoq.com/news/2014/01/IDC-software-developers
13. http://goo.gl/biFtVT
14. http://goo.gl/y32fsk
15. http://goo.gl/c5KzeI
16. http://goo.gl/pW1mTO

Des *développeurs* construisent des applications pour de multiples raisons comme par exemple pour gagner de l'argent, vendre du matériel associé, devenir célèbre, s'amuser ou aider les autres. Pour les petits développeurs, qui sont sans aucun doute les plus nombreux, le coût de la publicité et de la distribution de leurs logiciels sont prohibitifs que leur activité soit bénévole ou pas car les marges sont faibles lors de la vente de logiciel. Très peu de logiciels vont rencontrer un succès massif qui rapportera beaucoup d'argent et le succès est difficile à prévoir. Dans ce cadre, ils préfèrent se reposer sur des distributeurs établis qui leur offrent un canal de distribution établi et qui a la confiance des utilisateurs. Parmi de tels distributeurs, on peut citer par exemple, la distribution Debian et l'ensemble des paquets pour Linux qu'elle distribue ou Google et son magasin d'application pour Android, le *playstore* [17]. En échange, le développeur devra s'accommoder des exigences du distributeur en terme de délais d'approbation, de contraintes de qualité sur le produit, de formats de description, de prix du service... Notons immédiatement deux conséquences de la forte segmentation du marché des distributeurs en terme de plate-formes :

- Un développeur qui souhaite diffuser un logiciel sur plusieurs plates-formes va devoir gérer la compatibilité de son logiciel avec l'ensemble des distributeurs visés et maintenir toutes ses versions spécifiques.
- Un développeur qui réalise une application distribuée multi-plate-forme va devoir en plus gérer la synchronisation de ses différents canaux de distribution.

Un *distributeur* maintient un dépôt centralisé qui va contenir tous les logiciels qu'il distribue. Il va recevoir les logiciels et les vérifier pour s'assurer qu'ils respectent les contraintes qu'il a affiché aussi bien en terme de contenu, de forme d'empaquetage que des données de description [18]. Une fois le logiciel approuvé, il est catalogué et mis à disposition du client final. Ce service peut être facturé au développeur et / ou au client mais parfois il reste gratuit. Ainsi, Apple vend les logiciels de son *AppStore* aux clients et reverse au développeur uniquement 70 % du montant, la différence étant censée couvrir le prix de distribution. À l'opposé, Debian ou Ubuntu, offrent leurs services de manière gratuite. Certains dépôts permettent à un développeur d'exprimer des dépendances comme par exemple les dépôts de paquets Linux. Le mainteneur du dépôt doit alors également garantir un niveau global de qualité dans lequel les applications distribuées n'ont pas de contraintes insatisfiables par exemple. On parle alors de maintenir la bonne santé (*healthiness*) du dépôt [P76]. Cette tâche est coûteuse aussi bien en temps qu'en argent. Pour éviter une explosion du coût de cette validation, les dépôts de paquets Linux comme Debian limitent le nombre de paquets officiellement distribués [19]. Elle repose également sur une communauté d'*empaqueteurs*, ce qui rend la procédure de soumission / approbation parfois longue et donc fait que les logiciels distribués sont en retard par rapport à ce que le développeur à réalisé (de 10 à 30 semaines voire parfois 100 semaines d'après `http://oswatershed.org`). Les dépôts commerciaux font souvent un choix inverse, ils imposent des applications auto-contenues et peuvent donc plus facilement distribuer plus d'un million d'applications. Les développeurs doivent alors produire des applications plus simples ou bien embarquer dans toutes leurs applications les bibliothèques ou fonctionnalités annexes dont elles ont besoin. Les

17. On peut également citer des producteurs de matériel comme Apple ou Blackberry, des opérateurs réseaux comme Orange, des fournisseurs de services comme Facebook ou des fournisseurs de logiciels comme les fondations Apache ou Mozilla et Adobe.

18. Parfois, ces contraintes sont exprimées dans un langage tellement peu précis qu'il est difficile de savoir à l'avance si une application sera acceptée ou non.

19. Actuellement 37 500 pour Debian à comparer au 1,3 millions d'applications du *playstore* ou de l'*AppStore*.

fonctionnalités de très haut niveau qui sont communes à de nombreuses applications doivent alors être fournies par la plate-forme pour éviter que les applications deviennent obèses (les bibliothèques graphiques ou de localisation par exemple). En contrepartie, le développeur est contraint d'utiliser les bibliothèques de la plate-forme ce qui accentue encore la divergence du code des applications qui veulent être multi-plate-forme. Il faut, enfin, tenir compte de la validation des applications par les distributeurs. Il n'est pas toujours simple pour un développeur de savoir si son application va être acceptée sur un dépôt. En effet, certains dépôts ont une pratique de filtrage qui peut être considérée parfois comme de la censure [20].

Pour la distribution, les distributeurs réutilisent les techniques usuelles de diffusion de contenu à large échelle. Ainsi, les distributeurs qui le peuvent reposent sur les services de CDN (*Content Delivery Networks*) comme Akamaï qui possède des infrastructures de distribution de contenu un peu partout sur terre. Pour ceux qui ont moins de moyens, la technique des miroirs de diffusion répliqués un peu partout peut suffire. Par contre, ces stratégies ne tiennent pas vraiment compte de l'état du réseau et des autres ressources réseaux comme les opérateurs (fixes et mobiles) ou les ressources locales au réseau. Ainsi, un particulier possédant plusieurs machines devra en général télécharger plusieurs fois l'application depuis chacun des équipements. Avec une politique de miroirs ouverte, Debian lutte un peu contre cela, en offrant la possibilité d'héberger un miroir du dépôt chez soi. Cette approche est réaliste pour une petite structure disposant de spécialistes de l'informatique, elle semble moins crédible pour les particuliers qui dans le cadre de l'internet des objets auront bientôt sans doute un grand nombre d'équipements nécessitant du logiciel. De plus, la diffusion des logiciels reposant quasiment uniquement sur des actions initiées côté client, les transferts ne peuvent être qu'*unicast* (diffusion à un unique client). Avec des stratégies qui permettraient de pousser au plus près des clients les logiciels et leurs mises à jour, on pourrait passer à du transfert *multicast* voire *broadcast* qui réduirait considérablement la consommation de bande passante (surtout lors des pics de mise à jour majeure).

Enfin, les *utilisateurs* doivent assumer le rôle d'administrateurs de leurs équipements. Ils doivent choisir leurs logiciels et les versions à utiliser, réaliser les opérations d'installation, choisir lors d'une mise à jour s'il faut la faire ou non. De plus, comme déjà dit, ces opérations doivent être réalisées sur chacun des équipements et répétées régulièrement. De plus, pour peu qu'il utilise diverses plates-formes, il faut qu'il assure manuellement la synchronisation de toutes ces opérations. Au final, le système de distribution actuel met l'utilisateur au centre du processus alors qu'il n'est pas forcément compétent. Ainsi, par exemple, un utilisateur peut prendre une mauvaise décision sans forcément s'en rendre compte. Ainsi, il pourrait repousser à plus tard une mise à jour critique car elle lui impose de nombreuses opérations sur de multiples équipements qu'il possède. Si un usager a un contrat de paiement à l'usage pour le téléchargement, il pourra également hésiter face à des mises à jour.

Dans ce cadre, l'architecture généralement adoptée est présentée figure 4.10 page suivante. Dans cette architecture, il y trois formes de nœuds qui correspondent aux trois rôles identifiés précédemment. Ces trois nœuds interagissent à travers la notion de paquet (ici au centre de la figure). Ce paquet est accompagné de méta-données pour le décrire. Il est produit (1) par le développeur avec une partie des méta-données (par exemple la description, l'auteur ou les dépendances). Une fois produit, le paquet est soumis (2) à un distributeur qui le certifie (3) en ajoutant des méta-données (comme par exemple un certificat), le publie (4) puis éventuellement notifie (5) les utilisateurs. Finalement, les utilisateurs téléchargent (6) puis installent le logiciel (7).

20. http://www.southsearepublic.org/article/2438/read/apples_arbitrary_app_store_process/

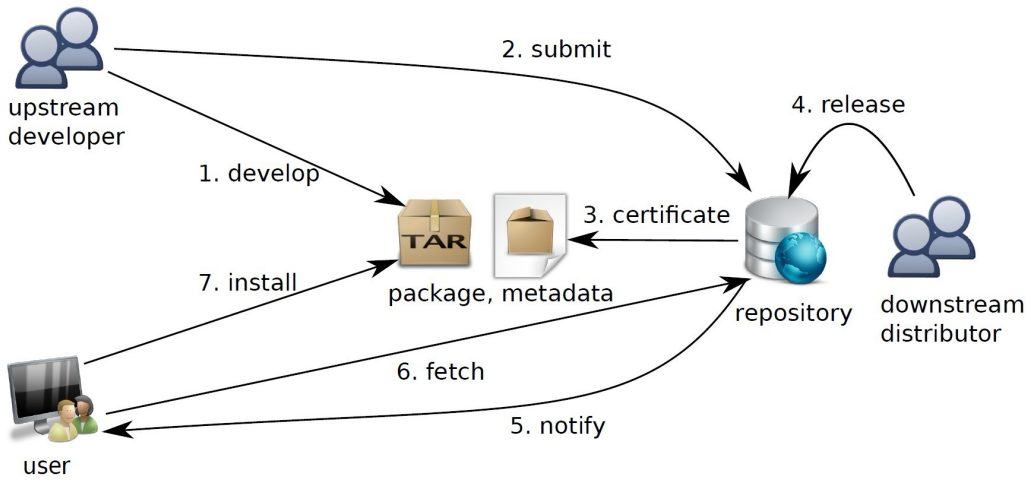


Figure 4.10 – L'architecture typique de distribution de logiciels à base de dépôt centralisé

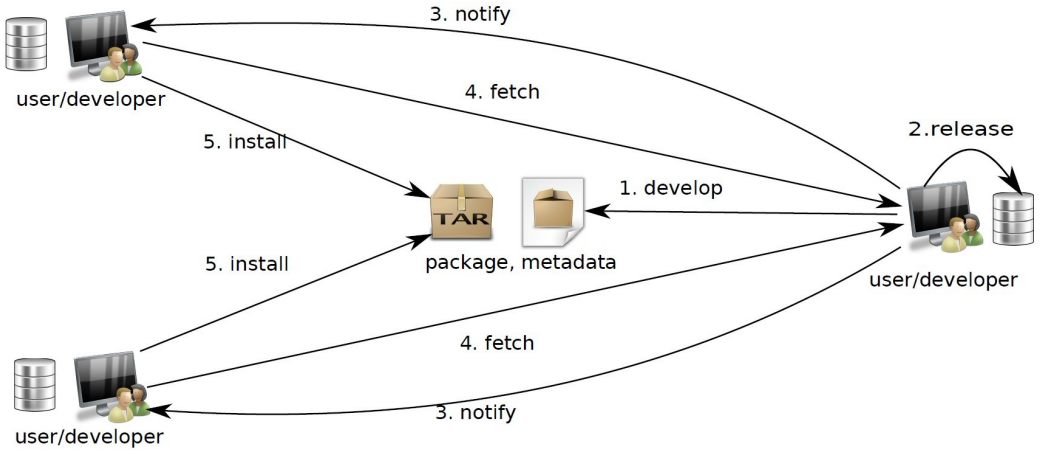


Figure 4.11 – L'architecture décentralisée proposée pour distribuer les logiciels

Cette section a présenté l'architecture actuelle de distribution de logiciels et résumé sa principale difficulté : le dépôt central. Ce dépôt assume de manière centralisée les rôles de :

1. espace de communication entre les développeurs et le distributeur (livraison),
2. espace de vérification et certification pour le distributeur,
3. espace de diffusion / publicité / notification par le distributeur,
4. espace de téléchargement pour les utilisateurs.

Ainsi, elle permet de montrer que des changements sont nécessaires dans cette architecture centralisée de distribution. Nous proposons donc de construire une architecture décentralisée de distribution de logiciels comme présenté dans la section suivante.

4.2.2 Vers un déploiement pair-à-pair

Notre proposition, `dpan`, suit une architecture pair-à-pair dans laquelle tous les nœuds sont de même nature. Ils peuvent donc potentiellement assumer les trois rôles nécessaires au déploiement d'un logiciel. Ainsi dans la figure 4.11 page ci-contre, chaque nœud est un dépôt dans lequel peut être publié (2) un logiciel, ce dépôt notifie (3) les autres dépôts de cette publication. Les nœuds interessés peuvent alors télécharger (4) le paquet. Ainsi, l'idée est de construire un réseau pair-à-pair de distribution de logiciels. Les échanges de données (notifications et fichiers) sont délégués au réseau pair-à-pair dont l'architecture est plus adaptée à la montée en charge qu'un dépôt centralisé. La construction d'un tel réseau soulève de nombreuses questions ouvertes que nous avons identifiées :

Comment `dpan` peut prendre en compte l'hétérogénéité des pairs ? Les nœuds sont des appareils très hétérogènes en terme de capacité. Un nœud peut être un serveur, un ordinateur personnel, un téléphone portable, un boitier TV, un capteur de température ou plus généralement, n'importe quel objet programmable de l'internet des objets. Chaque pair a, par exemple, des capacités de stockage, de calcul et d'accès réseau différentes. Un serveur d'entreprise pourra stocker un grand nombre de paquets alors qu'un objet de faible capacité devra sans doute déléguer à un pair une bonne partie de ses traitements (par exemple, la résolution des dépendances).

Comment permettre une évolution libre des paquets ? Tout pair doit pouvoir publier quand il le veut un nouveau paquet. L'identification des paquets doit rester unique et ne peut plus être garantie par le gestionnaire du dépôt. Il faut alors pouvoir gérer les différentes versions d'un paquet. La gestion usuelle à base d'entiers, de dates ou de combinaisons de ces éléments ne peut plus être utilisée. Enfin, le producteur du paquet doit pouvoir indiquer les dépendances de son paquet pour que les utilisateurs puissent l'installer et l'utiliser. Il convient de proposer un mécanisme qui permet de s'assurer de la complétude de la description des dépendances. En effet, on peut penser que dans un cadre d'utilisation de `dpan`, les plates-formes des utilisateurs vont être moins uniforme que maintenant. Par exemple, Nix [21] [P35] permet automatiquement de récolter et garantir les dépendances d'un paquet que l'on produit. Notons qu'il est également important de permettre l'installation de plusieurs versions d'un même paquet (ce que Nix supporte).

Comment rechercher et notifier au sein de `dpan` ? Les paquets sont conservés dans le nuage formé des différents pairs. La recherche et le téléchargement d'un paquet doit donc reposer sur l'infrastructure pair-à-pair. Celle-ci doit également assurer une répartition du stockage en fonction de la popularité des paquets tout en offrant des garanties de pérennité pour les paquets rarement demandés. Ceci doit prendre en compte le fait que le réseau des pairs est dynamique, des nœuds peuvent disparaître à tout moment. Ainsi, si la capacité d'un pair le permet, un paquet ne sera pas forcément effacé après installation pour pouvoir servir pour d'autres pairs. En l'absence de dépôt central, la recherche et la notification doivent utiliser de nouvelles approches reposant sur la communauté `dpan`.

Quel mécanisme de confiance au sein de `dpan` ? Comme chaque pair peut produire un paquet, il faut être capable d'identifier les paquets dignes de confiance. Il faut ainsi que les paquets soit signés pour qu'un pair puisse être sûr de l'origine d'un paquet. Des mécanismes de certification doivent être mis en place pour offrir des garanties. Les méta-données d'un paquet devront donc contenir des certificats. Ainsi, les distributeurs

21. https://nixos.org/nix

pourront continuer de produire des distributions garantissant un certain niveau de qualité en vérifiant les paquets circulant dans `dpan`. Par exemple, un projet comme Debian pourrait choisir des paquets, les vérifier, valider leurs dépendances puis y apposer son certificat. Il faut bien sûr que les certificats ne puissent pas être forgés par un pair mal intentionné. Ainsi, certains pairs pourraient fournir des dépôts de paquets certifiés en reposant sur les infrastructures actuelles. Un utilisateur ayant confiance en Apple pourrait continuer d'utiliser son *AppStore* qui ne serait alors qu'un pair comme un autre de `dpan` qui n'aurait donc aucune autorité sur les autres pairs. Notons que des mécanismes de DRM (*Digital Right Management*) seraient sans doute nécessaires pour les paquets propriétaires reposant sur des mécanismes de vente ou de licence.

Comment mettre à jour un paquet ? Chaque paquet doit pouvoir être rattaché à son histoire de façon à permettre la notion de mise à jour. Ainsi, l'ensemble des paquets doit être ordonné. Dans un système centralisé, le dépôt crée cet ordre souvent sur la base des versions mais aussi parfois en utilisant des méta-données [22]. De plus, cet ordre est total, il est toujours possible de savoir comment deux versions d'un paquet sont en relation. Dans un système décentralisé cet ordre total n'existe plus. Enfin, chaque pair doit pouvoir comparer des paquets pour savoir s'ils sont des versions différentes d'un même logiciel et l'ordre de ses versions. Pour cela, les méta-données doivent être enrichies par l'historique des paquets.

Du fait de l'abandon de Xu Zhang au bout d'un an et demi, seulement une partie de ses questions ont pu être explorées. Les sous-sections suivantes présentent ses premiers résultats.

4.2.3 Une distribution pair-à-pair pour quels usages ?

La figure 4.12 page suivante illustre quelques uns des cas d'utilisation que nous avons imaginés pour `dpan`.

Comme déjà présenté, un pair développeur (celui au nord dans la figure) peut décider de *publier* un paquet avec ses méta-données (metadata1 dans la figure). Il peut ensuite *notifier* ses voisins qui peuvent décider de le *télécharger*. Son voisin de gauche *modifie* les méta-données (en metadata2) et *re-publie* le paquet. Il *notifie* son voisin de sa nouvelle version du paquet. Les quatre utilisateurs du bas *installent* le logiciel, les trois de droite *installent* le paquet original, celui de gauche le paquet modifié. Celui du bas au centre ne réussit pas à l'installer et le signale à son voisin. Celui-ci va *voter* pour le paquet et ainsi indiquer que son installation peut échouer. Nous avons aussi ajouté l'action de *suivre* qui permet de connaître toutes les installations, mises à jour et désinstallations faites sur une machine. Ainsi, deux cas peuvent se présenter, la même opération est réalisée automatiquement sur la machine qui suit ou, selon une option de configuration, le système propose à l'utilisateur d'effectuer l'opération.

Ainsi, au-dela des opérations classiques des systèmes de distribution de logiciels, des fonctionnalités de l'ordre de communauté permettraient aux utilisateurs de s'entraider. La popularité des paquets, la fréquence des difficultés d'installation et les avis des utilisateurs contribuent à aider les utilisateurs dans leur recherche de logiciel. Cela permet aussi de leur donner confiance dans des paquets qui peuvent avoir été créés par des particuliers. Notons que le fait de suivre des développeurs ou de suivre les opérations sur une machine permet de faciliter la décision d'installation ou de mise à jour. La dernière fonctionnalité permet également à un utilisateur qui gère plusieurs machines de faire ses opérations de déploiement sur une unique machine, les

22. Dans Debian, il est possible d'indiquer qu'un paquet remplace un autre paquet.

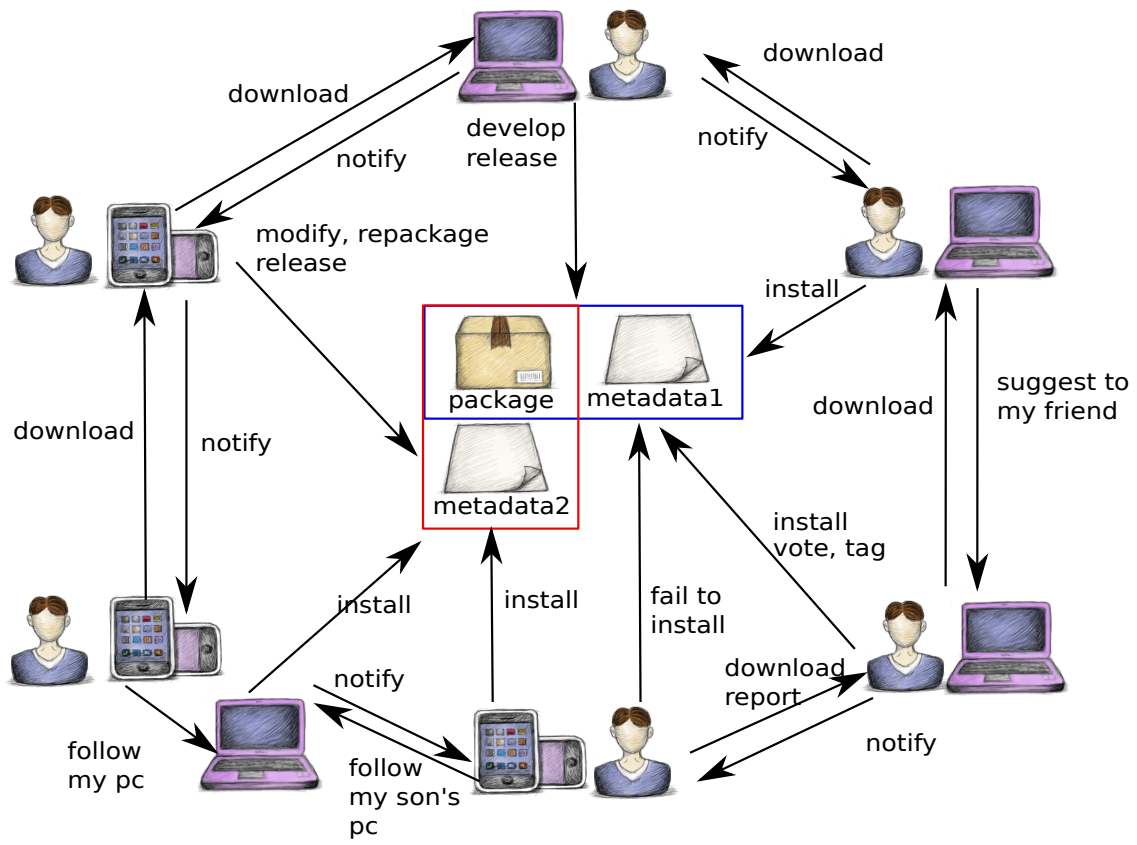


Figure 4.12 – Des utilisations possibles de `dpan`

autres machines pouvant se mettre à jour automatiquement si elles la suivent. On pourrait imaginer que des pairs gèrent d'autres machines que les leurs en utilisant des fonctionnalités pour déployer à distance.

Il est difficile d'anticiper toutes les utilisations qui pourrait être faites de `dpan`. Peut être que les différentes communautés existantes vont continuer à fonctionner comme maintenant. Nous soupçonnons quand même que les relations entre pairs vont changer. De nouvelles communautés de pratique apparaîtront sans doute. Le succès des différentes plates-formes collaboratives suggère qu'il est difficile d'anticiper les usages que les utilisateurs en feront. On pourrait voir apparaître de nombreux *empaqueteurs* spécialisés dans la certification et la diffusion de contenu. Des robots de fusion de paquets ou de construction de paquets contenant un paquet et ses dépendances pourraient également apparaître de façon à simplifier la vie des utilisateurs. Des experts pourraient conseiller les utilisateurs et leur *pousser*[23] des actions de déploiement...

4.2.4 Un format de méta-donnée pour `dpan`

Le but de cette sous-section est de présenter le format de méta-donnée que nous proposons dans le cadre de `dpan`.

23. Ici, pousser est utilisé pour l'anglais *push*.

Identification d'un paquet Un paquet contient un ensemble de fichiers et de données nécessaires à l'exécution d'une application. Il est identifié par une paire (fid, vid) où fid identifie la famille à laquelle le paquet se rattache et pid est un identifiant unique au sein de cette famille. La notion de famille a pour but d'indiquer qu'un ensemble de paquets fournissent le même logiciel. L'identifiant de version peut prendre plusieurs formes. Dans nos travaux, nous avons choisi d'utiliser une clé de contrôle suffisamment longue du contenu du paquet [24].

L'histoire d'un paquet Pour gérer la notion de mise à jour, il est nécessaire de disposer de l'histoire du paquet, c'est-à-dire l'ensemble des paquets de sa famille duquel il est un descendant. Pour cela, on utilise la relation *remplace* qui est définie par celui qui publie un paquet. Un paquet p_1 *remplace* un paquet p_2 est noté $p_1 \succ p_2$. Un paquet peut avoir plusieurs successeurs publiés simultanément par de nombreux pairs. L'histoire globale d'une famille de paquets pourrait grossir de manière incontrôlée. Pour permettre de diminuer cette potentielle explosion de versions, il est possible de *fusionner* plusieurs paquets en déclarant un paquet comme successeur de plusieurs paquets. Cela conduit a une histoire des paquets qui est un graphe dirigé acyclique. Notons que la relation *remplace* peut être définie entre deux paquets de familles (initialement différentes), cela conduit à une sorte de fusion des familles. Plus précisément, l'*histoire* d'un paquet p est le graphe $G^p = (\{p' \mid p \succ^* p'\}, \{(p_1, p_2) \mid p_2 \succ p_1\})$. On dit que p' est un *ancêtre* de p si et seulement si $p' \in G^p$. Notons que l'histoire fournit un ordre qui n'est pas total même si on le restreint à une famille de paquets. Cela signifie que l'on peut rencontrer deux paquets p_1 et p_2 de même famille ($p_1.fid = p_2.fid$ [25]) qui sont incomparables ($p_1 \notin G^{p_2} \wedge p_2 \notin G^{p_1}$). Pour un utilisateur, la notification de mise à jour et sa réalisation se base sur l'ordre induit de l'histoire de chaque paquet. Ainsi, si un utilisateur reçoit la notification d'existence d'un paquet p, il est considéré comme une mise à jour uniquement s'il existe un paquet p' sur la machine de l'utilisateur qui est dans l'histoire de p ($p' \in G^p$).

Les méta-données Les méta-données d'un paquet vont contenir les informations usuelles de description du paquet, ses éléments de configuration et des informations de sécurité. En réduisant la liste au minimum nécessaire [26], on obtient trois blocs :

- Le bloc de configuration pour les informations concernant le déploiement :
  - *l'histoire* : le graphe présenté précédemment ;
  - *les dépendances* : l'ensemble des paquets qui sont nécessaires mais aussi les conflits. Plus généralement, il est possible d'utiliser le langage de dépendance du chapitre 2 page 5.
- Le bloc de description :
  - *des alias* : les identifiants des paquets ne sont pas lisibles par des humains (ce sont des résumés cryptographiques), il convient donc de fournir des noms que les utilisateurs pourront comprendre et utiliser ; attention, ici aucune unicité n'est requise, plusieurs paquets peuvent donc avoir le même nom ;
  - *une description textuelle* : une explication du contenu du paquet pour les utilisateurs ;
  - d'autres informations comme l'auteur, la date de création, des étiquettes, ...
- Un bloc de sécurité :

24. En l'occurrence, un hachage cryptographique MD5 de 128 bit.
25. On utilise une notation pointée : si $p = (a, b)$ alors $p.fid = a$ et $p.pid = b$.
26. L'usage nécessitera sans doute d'ajouter des informations complémentaires comme des icônes, des liens, ...

- *des permissions* : il peut être nécessaire pour consulter les données d'équipements et / ou utiliser leurs ressources de disposer de jetons d'autorisation ;
- *des certificats* : le paquet peut avoir été certifié correct et de qualité par des pairs (organismes reconnus comme Debian ou particuliers).
- *des signatures* : des auteurs du logiciel, du paquet, des méta-données ...
- *des sommes de contrôle* : pour garantir l'intégrité du paquet et de ses méta-données.

Résumé de l'histoire La taille de l'histoire d'un paquet croît constamment. La transmettre dans chacune des notifications, la stocker et rechercher s'il existe une mise à jour sont donc des opérations consommatrices de ressources (réseaux, de stockage et de calcul). Nous proposons, pour permettre un meilleur passage à l'échelle, d'utiliser un résumé de cette histoire qui est suffisant pour avoir une bonne probabilité de décider si c'est une mise à jour. Pour cela, nous utilisons la notion de *filtre de Bloom*[27]. Un filtre de Bloom $\mathcal{BF}_E$ est un tableau de booléens de taille fixe m qui permet d'indiquer la probabilité d'appartenance d'un élément x (d'un univers $\mathcal{U}$) à ensemble E (de taille n, $E \subset \mathcal{U}$). Pour cela, il utilise k fonctions de hachage indépendantes $h_1, \ldots, h_k$ qui, à un élément de $\mathcal{U}$ associe un entier de $[\![1, m]\!]$. Pour chaque entier i de $[\![1, m]\!]$, on a $\mathcal{BF}_E[i] = \exists x \in E, j \in [\![1, k]\!]\ (h_j(x) = i)$. Une fois connu l'ensemble des éléments (E) et choisies les fonctions de hachage, le filtre peut être calculé. Il permet alors de tester si un élément x de $\mathcal{U}$ est dans l'ensemble E. Pour cela, on utilise :

$$x \in E \stackrel{\text{def}}{=} \bigwedge_{i=1}^{k} \mathcal{BF}_E[h_i(x)]$$

Un filtre de Bloom indique avec certitude que l'élément n'est pas dans l'ensemble (pas de faux négatif). Il indique la présence avec une probabilité P que l'élément peut être présent dans l'ensemble (il peut y avoir des faux positifs). Cette probabilité P dépend du nombre de fonctions de hachage k, de la taille de l'ensemble n et de la taille du filtre choisie m. Elle est donnée dans [P29, P22] et est difficile à calculer car coûteuse. En revanche, la valeur précédemment considérée comme correcte fournit une approximation plus simple à utiliser : $\mathcal{P}_{false} = (1 - e^{-kn/m})^k$. D'après [P29], cette formule reste proche de la probabilité réelle (mais inférieure) pour des m de l'odre de 1024 bit. En l'utilisant, on peut calculer que pour k valant 6, le pourcentage de faux positifs est contenu sous les 1 % pour des ensembles de moins de 107 éléments avec m valant 1 ko. Ce qui est déjà utilisable sur les histoires actuelles. En passant à 32 ko, on monte à 3407 éléments. Les valeurs pertinentes à choisir dépendront bien sûr de la taille réelle que prendraient ces histoires mais on constate que l'on peut trouver des valeurs raisonnables de m qui fonctionnerait probablement avec la majorité des paquets.

Le test d'appartenance reposant sur un filtre de Bloom peut être un faux positif. C'est-à-dire, le pair pense qu'un *nouveau* paquet dont il vient de recevoir la notification est une mise à jour d'un des siens. Le système de mise à jour va alors télécharger le paquet en question. Il disposera ainsi de l'histoire du paquet et pourra effectuer un nouveau test d'appartenance qui lui est sûr. On peut alors s'apercevoir que le paquet en question n'était pas une mise à jour. Le coût des faux positifs correspond à la nouvelle vérification après téléchargement de la relation *remplace* et parfois au téléchargement d'un paquet inutile.

Modèle Au final, le modèle de paquet que nous proposons est présenté dans la figure 4.13 page suivante.

27. https://fr.wikipedia.org/wiki/Filtre_de_Bloom

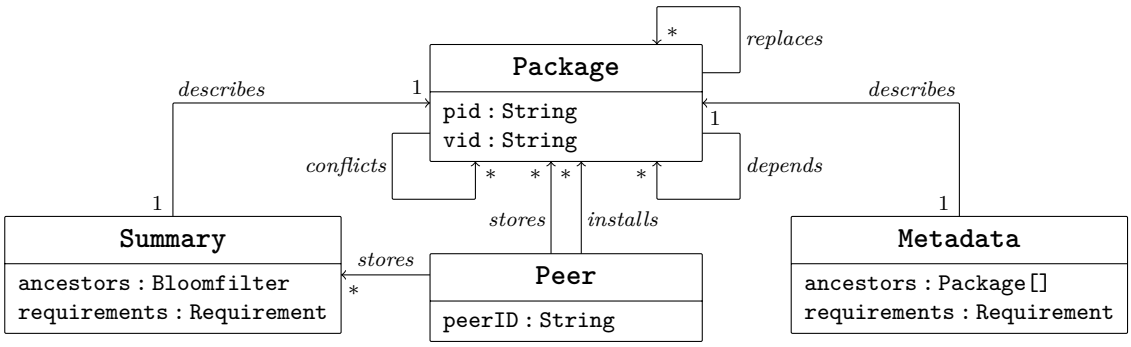


Figure 4.13 – Le modèle conceptuel de `dpan`

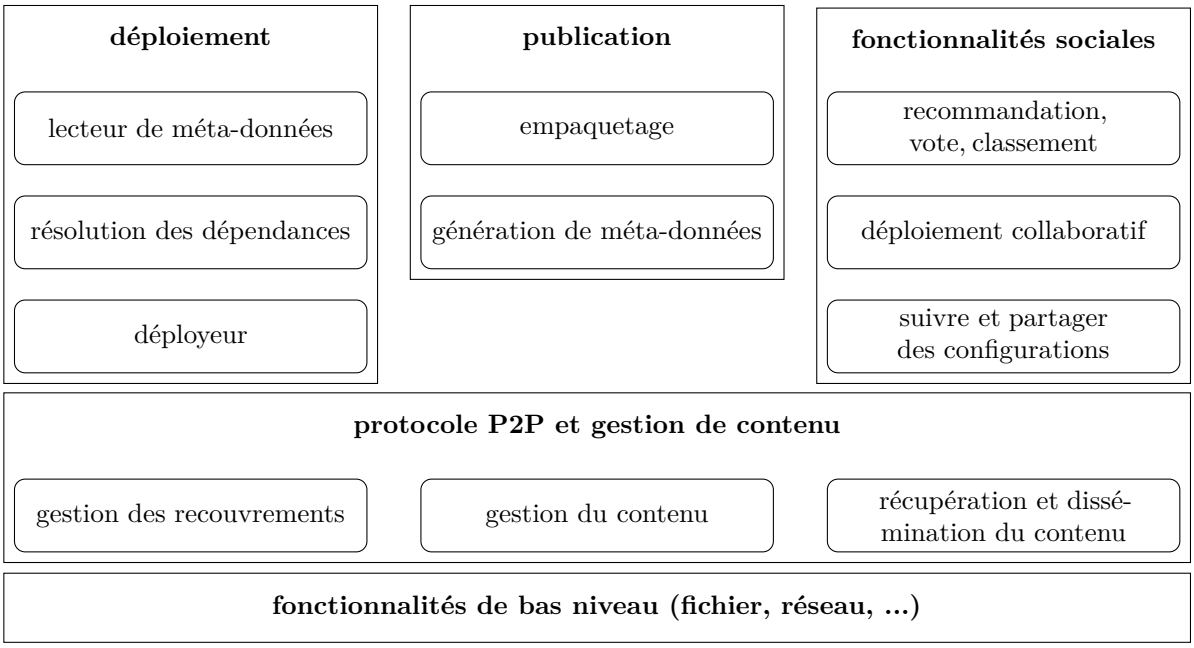


Figure 4.14 – Les fonctionnalités principales de `dpan`

4.2.5 Les fonctionnalités de `dpan`

Du point de vue de l'utilisateur, l'application doit fournir les fonctionnalités des outils actuels de déploiement et des magasins d'applications et les étendre. L'architecture générale de `dpan` est un ensemble de pairs identiques (en terme de rôle). Ces fonctionnalités, présentées dans la figure 4.14, peuvent être regroupées en cinq blocs :

1. Les **fonctionnalités de bas niveau** sur lequel le client `dpan` repose. Le client doit pouvoir accéder au système de fichier de son hôte pour y stocker ses informations. Il faut également que les pairs puissent communiquer, on a donc besoin de fonctionnalités réseaux de bas niveau. Ces fonctionnalités ne posent pas réellement de difficultés.

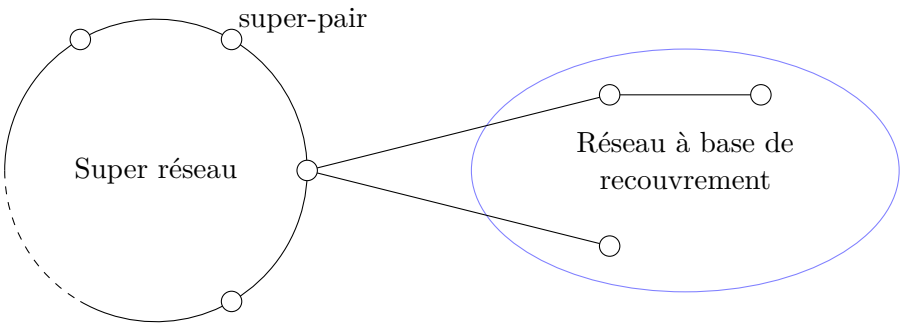


Figure 4.15 – L'architecture réseau de `dpan`

2. La gestion du **protocole pair-à-pair et du contenu** [28] permet de fournir toutes les opérations nécessaires au maintien du réseau pair-à-pair, au routage en son sein et à la distribution en son sein du contenu. Pour la gestion du contenu, il s'agit de gérer les stockage des paquets et de leurs méta-données au sein du réseau de pairs. Il faut ainsi garantir la fiabilité du contenu et si possible la rapidité d'accès. Il faut également fournir les opérations de recherche et téléchargement des paquets et de leurs méta-données. Il convient enfin de gérer la propagation des notifications. Le réseau pair-à-pair conçu repose sur la notion de recouvrements (*overlay*). Ainsi, chaque pair maintient des listes de voisins [29] avec lesquels il communique. La structure choisie pour `dpan` est une structure basée sur les versions des paquets. Ainsi, tous les pairs qui ont installé une certaine version d'un paquet sont dans le même groupe. De plus, chaque groupe maintient des liens avec les groupes qui ont installé une version que la leur remplace. On obtient une structure de réseau utilisant des recouvrements sémantiques hybrides qui permet un routage efficace des messages. Au final, les pairs vont être regroupés suivant une mesure de similarité des paquets qu'ils ont installé proche de [P9]. Enfin, pour permettre de trouver efficacement des paquets rares, il convient de maintenir une deuxième structure utilisant une table de hachage distribuée (DHT) qui indexe les différents paquets. Cette table ne sera pas distribuée sur tous les pairs mais uniquement sur un sous ensemble que nous appelons les super-pairs. Cela amène à l'architecture présentée figure 4.15. Chaque pair qui n'est pas super-pair est relié à (au moins) un super-pair. Ainsi, si la recherche du paquet n'aboutit pas dans son voisinage, il peut passer par le super-pair qui lui trouvera dans la DHT un pair qui possède le paquet.
3. L'**installateur** est le composant qui sera en charge des opérations de déploiement (installation, mise à jour et désinstallation). Pour cela, il doit lire les méta-données du paquet pour résoudre ses dépendances. Une fois les dépendances résolues (par exemple, en installant de nouveaux paquets), le déploiement peut être réalisé [30]. La sémantique précise de cet installateur n'a pas fait l'objet d'une étude très approfondie. Elle suit, dans notre prototype, la sémantique de distribution des paquets Debian. Ainsi, par exemple, l'installation se

28. Je ne rentre pas trop dans les détails de cette partie du projet `dpan` car ce n'était pas mon domaine de compétence mais celle de mon collègue Gwendal Simon qui co-encadrait ce travail.
29. Évidemment, pour des raisons de passage à l'échelle, aucun pair ne connait tout son groupe (sauf s'il est vraiment très petit).
30. On retrouve une séparation entre les préoccupations de résolution et de réalisation du déploiement sur le modèle de la proposition du chapitre 2.

fait dans l'arborescence système, elle est donc destructive : installer un paquet de même version qu'un paquet déjà installé le remplace. Or, comme déjà signalé précédemment, cette sémantique ne convient pas pour le futur `dpan` dans lequel il est crucial de pouvoir installer un paquet dans plusieurs versions.

4. La gestion de la **publication** contiendra des fonctions qui permettent à un utilisateur de contribuer facilement même s'il n'est pas un développeur et qu'il souhaite modifier un paquet par exemple, pour traduire sa description ou rajouter des recommandations d'installation et d'usage. Cette partie n'a pas pu être abordée dans le cadre du projet et reste donc ouverte.

5. Les **fonctionnalités sociales** ont pour but de fournir des opérations pour développer des communautés autour du déploiement. Nous avions prévu d'explorer trois formes de communautés : celles autour de paquets, celles autour de configurations (ensemble de paquets) et celles pour la gestion collaborative de machines. La seule qui a vraiment été explorée est celle portant sur l'ajout et la diffusion d'informations de recommandation, de vote et de classement sur les paquets. Dans ce cadre, nous avons proposé un processus de collecte des nombres de téléchargements, installations, désinstallations de paquet ; les certificateurs qui ont la confiance des pairs, les distributeurs utilisés et les utilisateurs suivis (utilisation de la fonction je *suis* les déploiements d'un pair). Nous collectons également des notes attribuées aux paquets suivant des critères comme par exemple l'utilité, la robustesse, l'ergonomie... À partir de ces données et des méta-données des paquets, il est possible d'attribuer une popularité aux paquets, leurs développeurs, leurs distributeurs et certificateurs. Cet algorithme de collecte repose sur la structure réseau de `dpan`. Dans chaque groupe du réseau de recouvrements, on collecte les statistiques sur les paquets, certains des pairs, les *collecteurs*, envoient ces informations vers le réseau des super-pairs, où l'information est consolidée. Elle est alors rediffusée du réseau des super-pairs vers les autres pairs. Notre structure peut être également utilisée pour la collecte d'avis même si nous n'avons pas expérimenté cet aspect.

4.2.6 Une simulation de `dpan`

Afin de valider nos premières idées, nous avons développé un prototype de simulation de `dpan`. Pour cela, nous avons développé en Java une partie des blocs fonctionnels de `dpan`, des classes pour simuler le comportement d'un pair ainsi que des programmes pour gérer une simulation répartie. Le principe de fonctionnement de ce simulateur est le suivant. Un ensemble de machines est choisi. Chaque machine va héberger un ou plusieurs pairs. Un pair s'exécute dans sa JVM, il suit un comportement décrit dans un fichier de configuration. Ce comportement consiste en une séquence d'instructions de déploiement ou de diffusion de paquets.

Pour simuler de manière assez réaliste le fonctionnement du déploiement, nous avons extrait depuis un des sites web de Debian [31], l'ensemble des 110 763 paquets publiés entre mars 2005 et mars 2011. Nous avons ensuite reconstruit leur histoire en suivant toutes leurs mises à jour [32]. Ces paquets ont en moyenne 9.18 versions par paquet pour un nombre variant de 1 à 302. Ensuite, nous avons transformé tous ces paquets pour suivre le format de méta-données de

31. http://snapshot.debian.org
32. Cette histoire n'est probablement pas très représentative de ce qui se passerait avec `dpan` car elle contient peu de divergence / fusion de paquet. Cela faisait partie des travaux suivants qui n'ont pas eu lieu de tenter de complexifier cette histoire.

`dpan`. Cela implique, l'attribution d'un identifiant par hachage du contenu, l'inclusion de son histoire, la transformation de toutes les dépendances pour qu'elles utilisent les identifiants de paquet avec version de `dpan` et le calcul du résumé de l'histoire de chaque paquet. On obtient ainsi une base de 1 016 804 paquets. Cette reconstruction a été coûteuse en terme de calcul puisqu'il a fallu une moyenne de 4 s par version de paquet soit 48 jours de calcul.

Afin de simuler la diversité des pairs, nous avons récupéré la popularité de chacun des paquets sur le site de Debian [33]. À partir de cette popularité, chaque installation initiale d'un pair est construite de la façon suivante. Tout d'abord, on construit un ensemble de paquets de manière aléatoire. La probabilité de prendre un paquet dépend bien sûr de sa popularité réelle. Une fois ce cœur de paquets déterminé, on choisit une version pour chaque paquet de manière également aléatoire avec une probabilité plus forte pour les versions récentes mais pas trop. Enfin, on détermine à partir des dépendances de chacun des paquets, l'ensemble des paquets constituant réellement l'installation [34]. Les pairs contiennent également initialement un ensemble aléatoire de (versions de) paquets supplémentaires.

Cette infrastructure a été utilisée principalement pour valider la collecte des popularités et la faisabilité générale de l'architecture de `dpan`. Le code Java consiste en environ 5500 lignes de Java pour le client `dpan` et 700 lignes de Java et 1800 lignes de Python pour la conversion des paquets.

4.3 Bilan

Comme indiqué en introduction de ce chapitre, ces travaux ne sont pas complètement aboutis. Ils contribuent néanmoins en posant des questions qui me semblent fondamentales sur le déploiement du logiciel :

- Comment faire en sorte que le déploiement d'un logiciel soit plus flexible pour s'adapter à une large variété d'environnements tout en offrant des garanties ? Dans ce cadre, il me semble important de disposer de langages offrant divers degrés d'abstraction permettant aux développeurs et déployeurs de décrire de manière précise les exigences qu'un logiciel doit satisfaire une fois déployé. Il faut aussi que l'infrastructure de déploiement soit suffisamment ouverte pour permettre à tout le monde de contribuer.
- Comment faire pour aider les utilisateurs dans leurs opérations de déploiement ? L'emergence de communautés et le lien fort entre ces communautés et les infrastructure de déploiement devrait permettre d'aider les utilisateurs. En effet, dans un monde totalement digital dont la structure matérielle mais aussi logicielle devient de plus en plus complexe, il est crucial de faciliter la manipulation de logiciels.

Une partie délicate dans ce type de projet de recherche est la quantité de standards, systèmes et logiciels avec il faut s'accommoder. Ainsi, toute expérimentation devient coûteuse car il faut déployer des plates-formes et construire des prototypes. En particulier, lors de mes travaux dans le cadre du déploiement décentralisé, il n'existait pas, à notre connaissance, d'outils permettant de simuler l'exécution d'une telle plate-forme. Malgré ces difficultés, ces questions me semblent centrales et je ne compte pas les abandonner.

33. http://popcon.debian.org
34. Ce processus peut échouer si le choix inclus un conflit mais alors recommence à nouveau tout le processus de construction.

Chapitre 5

La mise à jour à chaud de logiciel

Ce chapitre résume les travaux que j'ai réalisé dans le domaine de la mise à jour à chaud de logiciel. La mise à jour à chaud sera appelé DSU pour *Dynamic Software Update* par la suite.

Le DSU désigne l'ensemble des techniques pour mettre à jour ou étendre un système logiciel sans en interrompre l'exécution et sans (trop) dégrader les services qu'il offre. L'utilisation de ces techniques est vitale dès lors que le système logiciel a un rôle critique, comme par exemple pour les systèmes de télécommunications et les systèmes embarqués. Ici, il ne s'agit plus de gérer les dépendances et le déploiement de la mise à jour mais bien d'offrir des plates-formes d'exécution qui réalisent cette mise à jour à chaud.

Ces travaux de recherche ont débuté dans le cadre du projet ANR SPaCIFY où mon équipe avait en charge la réalisation d'une plate-forme d'exécution pour les logiciels de contrôle des satellites. Parmi les services que devait contenir cette plate-forme, l'accent a été mis, très tôt, sur le DSU du fait de son importance pour un satellite. Ainsi, depuis 2007, ce sujet est au cœur de mes préoccupations.

Ces travaux ont été publiés dans diverses conférences et revues [10, 28, 30, 3, 14, 1, 36, 15, 37]. Les principales contributions de cette activité de recherche sont :

- Une méthode de mise à jour pour les satellites.
- La proposition d'une technique de DSU permettant de toujours pouvoir mettre à jour immédiatement. Cette technique repose sur la notion de continuation et l'utilisation d'opérations de haut niveau pour manipuler cette continuation (la parcourir, en retirer ou y ajouter des bouts, extraire des données de l'application).
- ReCaml, une extension du langage Caml pour le support du DSU avec une syntaxe pour le support de la manipulation des continuations.
- Un système de type statique qui garantit une manipulation sûre des continuations et des données qu'elles contiennent. La correction de ce système a été mécanisée en Coq principalement par Jérémy Buisson.
- Une bibliothèque Python Pymoult pour permettre l'implantation aisée de nouveaux mécanismes de mise à jour à chaud. Cette plate-forme doit permettre à terme de fournir un environnement d'évaluation et de combinaison de techniques de DSU. Ces travaux constituent le cœur de la thèse de Sébastien Martinez qui devrait bientôt soutenir.
- Un modèle de composant Pycots réalisé en Python et sa traduction en un modèle Coq. Un processus outillé qui supporte des reconfigurations architecturales prouvées d'applications Pycots. Ce processus repose sur une couche d'introspection de Pycots qui permet d'extraire

une description Coqcots d'une application en cours d'exécution. Le développeur produit ensuite en Coq sa reconfiguration ainsi que sa preuve. Le code de reconfiguration est ensuite extrait et appliqué à l'application à l'aide de Pymoult.

5.1 Les problématiques de la mise à jour à chaud

5.1.1 Généralités

Les logiciels sont des entités complexes construites par assemblage d'éléments plus simples (comme par exemple des objets, modules, composants, ...). La composition des sous-éléments définit la *structure* du logiciel qui les contient. De plus, une application fournit des services (parfois appelés fonctionnalités) en utilisant des ressources fournies par l'environnement. Ce que l'on appelera la *sémantique* de l'application. Enfin, quand elle s'exécute, une application a un *état* (considéré comme vide si l'application ne s'exécute pas).

À chaque instant, une application a une *configuration*, le triplet composé de sa structure, sa sémantique et son état. *Reconfigurer* consiste à changer la configuration d'une application depuis l'extérieur (l'application subit une reconfiguration). Parfois, les reconfigurations sont appelées des mises à jour (*updates* en anglais) ou des mises à niveau (*upgrades* en anglais plus utilisé que le terme français). Dans [P14], Barton distingue les reconfigurations qui n'ont pas d'effet sur l'environnement de l'application (*updates*) de celles affectant l'environnement (*upgrades*).

Les reconfigurations permettent de corriger des *bugs* mais également de faire face à de nouvelles exigences en ajoutant de nouveaux services à une application. Cela permet également d'adapter une application au changement de son environnement pendant son exécution (par exemple, lui supprimer un service devenu inutile).

Les reconfigurations sont parfois indispensables lorsque l'arrêt ou la suspension d'un logiciel est trop coûteux voire n'est pas possible. Par exemple, arrêter un équipement de cœur de réseau n'est pas possible pour un opérateur de télécommunication car l'effet d'une telle opération est difficile à prévoir du fait des montées en charge provoquées aux autres équipements. Un satellite, une fois en orbite, ne peut pas non plus être arrêté car il n'est alors plus contrôlable depuis le sol. Hicks [P58] cite aussi l'exemple du système de paiement de Visa qui utilisait vingt et un *mainframes* pour assurer trois mille transactions par seconde. Son code de cinquante millions de lignes était mis à jour toutes les vingt-six minutes en moyenne tout en devant être utilisable à 99.5 % (soit au maximum dix-huit secondes d'arrêt par heure) [P95]. Au delà de ces exemples, la reconfiguration à chaud offre un confort d'usage aux utilisateurs d'un système puisqu'elle diminue voire supprime les périodes pendant lesquelles le système ne peut pas être utilisé. Même un utilisateur standard a déjà été obligé de redémarrer le système d'exploitation de son ordinateur pour appliquer une mise à jour (qui en général n'arrive pas au bon moment...) alors que les « *reboots* sont pour le matériel pas pour le logiciel » comme défendu dans [P16]. Enfin, tout développeur rêve d'un environnement de travail où il peut modifier un logiciel sans arrêter son exécution et ainsi il n'a pas besoin à chaque modification de recommencer sa session de mise au point. Les environnements de développement de .NET et Java [P34] supportent déjà certaines modifications du code pendant une session de débogage.

Schématiquement, une application est composée de code (qui décrit à la fois sa structure et sa sémantique) et de données (qui traduisent son état). L'exécution d'une reconfiguration doit donc produire un nouveau code et des nouvelles données (à partir des données actuelles) et poursuivre l'exécution avec eux. Pour cela, un gestionnaire de reconfiguration doit s'exécuter simultanément

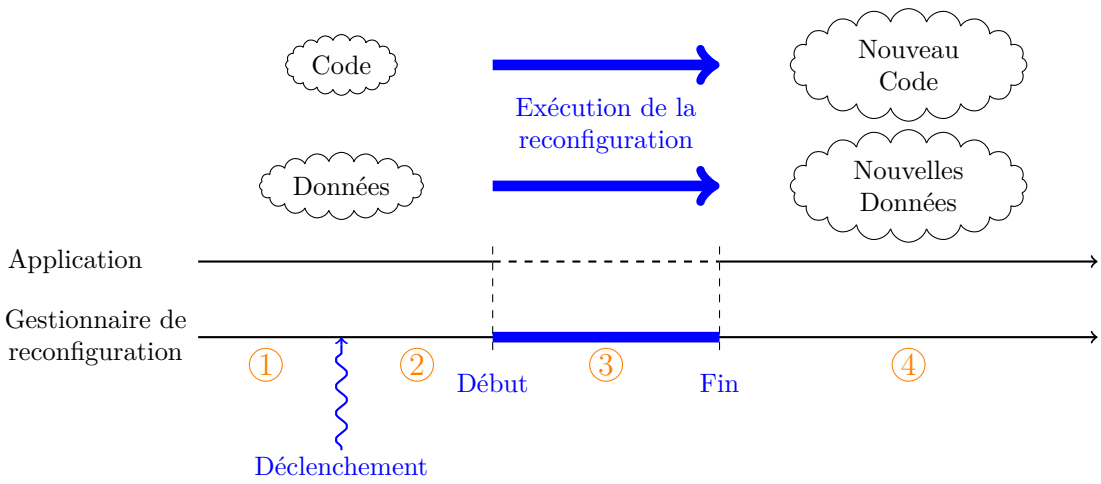


Figure 5.1 – La reconfiguration d'une application

à l'application [1]. Il est responsable de l'exécution de toutes les actions qui sont nécessaires à une reconfiguration. Il peut être générique, on parle alors d'*intergiciel de reconfiguration*, ou être dédié à une application. La figure 5.1 illustre cette description en représentant l'application et son gestionnaire.

L'exécution d'une reconfiguration suit quatre états :

1. *La reconfiguration a été spécifiée et empaquetée :* pour atteindre cet état le concepteur de la reconfiguration doit construire tous les artefacts nécessaires (modèles, codes, binaires, ...) en se basant sur la spécification de l'application et sur son état courant. La section 5.2 page 78 précisera une telle étape de conception pour les logiciels de vol des satellites.
2. *La reconfiguration a été déclenchée :* le gestionnaire de reconfiguration a reçu un signal lui demandant l'exécution d'une reconfiguration. La reconfiguration est généralement déclenchée par un opérateur ou un mécanisme extérieur mais l'application, elle-même, peut demander une reconfiguration. Il attend le moment approprié (une date, un état de l'application, un état du système, ...) pour la démarrer.
3. *La reconfiguration est en cours d'exécution :* le chargement des nouveaux éléments se fait pour permettre la construction du nouveau code et les nouvelles données sont produites à partir des anciennes. Comme suggéré sur la figure par des pointillés, l'application s'exécute éventuellement dans un mode dégradé (elle offre moins de services ou moins de qualité de service).
4. *La reconfiguration est terminée :* le gestionnaire a déterminé que l'exécution de la reconfiguration est terminée. Il a lancé le nouveau code avec les nouvelles données. Son rôle vis-à-vis de cette reconfiguration est terminé.

5.1.2 Les problématiques de la reconfiguration

Offrir un mécanisme de reconfiguration est difficile et pose des défis aux chercheurs et développeurs :

1. Le gestionnaire peut être une partie de l'application ou un composant ajouté. Il peut s'exécuter tout le temps ou uniquement de temps en temps. Comme nous le verrons dans la suite de ce chapitre l'univers des possibles est relativement grand.

Comment une reconfiguration est-elle conçue ? Une reconfiguration est en partie une application classique faite pour répondre à des besoins et donc doit suivre un processus de développement usuel. Cependant, sa conception est un peu plus complexe car elle doit tenir compte de l'existant. En effet, la réalisation d'une reconfiguration est fortement liée à l'application à reconfigurer. Enfin, un point délicat est celui de la prise en compte de l'état d'exécution de l'application. Il est possible de fixer l'état (ou les états) dans le(s)quel(s) l'application sera reconfigurée. Mais parfois il faut que le code de la reconfiguration s'adapte à l'état courant réel de l'application. Elle doit donc être conçue pour cela et prendre en compte les différentes possibilités.

Que contient un paquet de reconfiguration ? Il s'agit ici d'identifier les éléments qui doivent être communiqués au gestionnaire de reconfiguration pour qu'il puisse la déclencher. Différentes stratégies sont possibles. On peut, par exemple, embarquer des correctifs (*patches*) à appliquer au code en cours d'exécution. On peut aussi empaqueter directement le nouveau code ou enfin fournir un programme qui saura construire le nouveau code. Les mêmes stratégies s'appliquent pour la construction des nouvelles données. Enfin, la reconfiguration doit également embarquer une description de son moment d'application. Le contenu exact du paquet dépendra beaucoup des fonctionnalités du mécanisme de reconfiguration. Ainsi, par exemple, s'il intègre un interpréteur de script de reconfiguration, il suffit d'embarquer un script. Dans le cas contraire, il faut du code exécutable directement (un module compilé par exemple).

Comment une reconfiguration est-elle déclenchée ? Ensuite, il faut aborder la question de la façon dont est déclenchée la reconfiguration. En général, il s'agit d'un signal extérieur asynchrone par rapport à l'exécution de l'application. Ce signal peut être matériel ou logiciel, le gestionnaire devra être à son écoute. La source de ce signal peut être un opérateur humain par exemple dans le cas d'une correction de *bug*, l'administrateur déclenche l'application d'un correctif (à chaud). Si la reconfiguration doit répondre à un changement de contexte, l'application ou le gestionnaire doivent surveiller le contexte. Ils sont alors en charge du déclenchement de la mise à jour en cas de changement de contexte. Le mécanisme de reconfiguration doit donc fournir une interface pour le déclenchement qui doit contenir les différents mécanismes supportés.

Quand la reconfiguration est-elle exécutée ? Le paquet d'une reconfiguration contient une spécification des moments acceptables de reconfiguration. Le gestionnaire de reconfiguration doit alors calculer le moment auquel démarrer la reconfiguration d'après cette spécification. En général, on obtient soit une date soit un ensemble d'états que l'application doit atteindre. Il doit ensuite attendre ce moment. Cette attente peut être passive s'il se contente d'observer ou active s'il pousse l'application vers un certain état par exemple. Souvent, le gestionnaire doit également tenir compte de contraintes de l'environnement comme par exemple des échéances temps réel ou bien le fait que l'application doit continuer à fournir (une partie de) ses services.

Comment la reconfiguration est-elle exécutée ? Il faut ensuite, une fois le moment atteint, que le gestionnaire insère le code de la reconfiguration dans l'exécution et lui fournisse les ressources dont il a besoin (des *threads*, des données, ...). Le nouveau code doit parfois être calculé (par exemple dans le cas de l'utilisation d'un *patch*). Il faut, ensuite, le charger en mémoire suivant différentes techniques disponibles qui, en général, tournent autour du chargement dynamique de code et exigent de lier des éléments de l'ancien code et du nouveau (en anglais, on parle de *rebinding* ou *relinking*). Cette tâche

repose sur la structure de l'application (ses fonctions, objets, modules, composants, ...). Ensuite, il faut produire les nouvelles données en fonction des anciennes et les transmettre au nouveau code. Ces transformations sont souvent complexes lorsque les reconfigurations nécessitent des changements de type. En effet, l'ancien code et le nouveau peuvent avoir à cohabiter le temps de la reconfiguration et s'ils ne manipulent pas les mêmes types de donnée, il peut apparaître des erreurs de type. Notons, que cette transformation peut être longue (par exemple, transformer toutes les instances d'un type contenues dans le tas) et alors on adopte parfois une stratégie paresseuse dans laquelle la donnée n'est transformée que lorsqu'elle utilisée pour la première fois par le nouveau code. Enfin, les données d'une application sont parfois complexes à récupérer (elles peuvent être dans les registres, dans la pile, dans le tas, dans des bases de données, ...).

Comment cacher l'exécution de la reconfiguration ? Ici, il s'agit d'assurer que l'application continue à rendre ses services, au moins partiellement. Le gestionnaire doit basculer l'application dans un état où elle peut être reconfigurée tout en continuant son exécution. Il y a obligatoirement une dégradation de service et un choix doit être fait sur la forme de cette dégradation. Certains services de l'application peuvent être suspendus ou alors la qualité des services peut être diminuée (par exemple, en acceptant moins de sollicitations ou en mettant plus de temps à répondre). Un des défis majeurs du domaine est de rendre cette dégradation la plus transparente possible pour les utilisateurs de l'application. La définition de cette transparence dépend essentiellement du domaine de l'application.

Comment assurer la correction d'une reconfiguration ? Pour reconfigurer une application, il faut tenir compte de trois spécifications : (a) celle de l'application avant sa reconfiguration, (b) celle valide pendant la reconfiguration et (c) celle de l'application après sa reconfiguration. (a) et (c) sont naturellement issues des exigences sur les deux versions de l'application (avant et après la reconfiguration). La vraie difficulté concerne (b) qui dépend de l'application, de la plate-forme d'exécution, du mécanisme de reconfiguration et des choix du concepteur de la reconfiguration. Cette spécification doit couvrir la diminution des services rendus par l'application ou de leurs qualités. Une fois spécifiée cette correction doit être assurée conjointement par la plate-forme de reconfiguration et le gestionnaire de reconfiguration. Si (b) ou (c) sont violées, le gestionnaire de reconfiguration doit interrompre la reconfiguration et l'annuler (dans la mesure du possible). Souvent, des techniques de transaction sont utilisées pour gérer les reconfigurations. Enfin, au-delà de cette vérification durant l'exécution, on peut proposer des analyses statiques qui permettraient de prouver tout ou partie de ces propriétés avant l'exécution de la reconfiguration.

5.1.3 Quelques propositions

De nombreuses techniques de reconfiguration ont été proposées. Ainsi, [P101] identifie environ cinquante propositions. D'autres états de l'art existent également [P81, P104, P91] pour le lecteur interessé. Ici, je vais me contenter de citer quelques uns des travaux les plus influents du domaine. Cette courte présentation est structurée en deux parties. Nous détaillons d'abord les travaux effectués dans le cadre des modèles de composant. Puis, nous présentons ensuite les outils de modification dynamique des liaisons (*dynamic rebinding*).

Reconfiguration d'applications à base de composants Lorsqu'une application est un assemblage de composants, il est possible de s'appuyer sur cette structure pour reconfigurer.

Une mise à jour consiste alors à ajouter [2] et/ou enlever des composants et des liaisons dans l'assemblage. Par exemple, dans le modèle de composants Fractal [P24], chaque composant fournit des interfaces de contrôle pour le restructurer. Dans OpenCOM [P32], les opérations de reconfiguration sont laissées à la charge de la plate-forme à travers une représentation reflexive des différents éléments de l'application (composant, interface et liaison). Pour éviter les interférences avec le reste de l'application, les composants affectés par une reconfiguration sont isolés. Leur exécution est suspendue ce qui permet d'échapper temporairement aux règles de validité des assemblages (les composants en questions peuvent être débranchés par exemple). Des propriétés telles que la *quiescence* [P66, P67] et la tranquillité [P112] déterminent quels sont les composants qui doivent être effectivement arrêtés en garantissant que les composants concernés par une mise à jour ne seront ni actifs ni activés pendant son déroulement. Pour cela, la quiescence impose de suspendre les composants cibles de la reconfiguration ainsi que leurs clients afin d'éviter qu'ils initient des requêtes vers un composant en cours de reconfiguration. Lorsque c'est possible, la tranquillité relâche cette exigence en permettant l'exécution des clients tant qu'ils ne font pas de requêtes vers les composants reconfigurés. Plus généralement, il s'agit de garantir qu'une reconfiguration se comporte comme une transaction insérée dans l'application, c'est-à-dire que les reconfigurations doivent être atomiques même si plusieurs composants sont concernés.

Modification dynamique des liaisons La modification de programme a été traitée de longue date par les systèmes réflexifs et par la méta-programmation. Ainsi, plusieurs environnements permettent de remplacer de manière atomique un élément d'un programme (ici, il s'agit, en général, d'une fonction, d'une classe ou d'un module). Par exemple, Erlang [P11] permet de remplacer le corps des fonctions exportées par un module. Les infrastructures de la machine virtuelle Java et de la CLR .NET permettent de redéfinir le corps des méthodes d'une classe. En Smalltalk, la méthode `become` permet de rediriger toutes les références vers un objet sur un autre objet [3] [P103, FAQ 5.14]. Plus largement dans les langages d'acteurs, la méthode `become` permet de modifier le comportement d'un acteur et donc de supporter des mises à jour si le nouveau comportement a été reçu de l'extérieur par l'acteur. Par exemple, le comportement suivant en pseudo-erlang :

```
beh_with_update() ->
receive
...
[update,Behavior] -> Behavior()
end.
```

permet à un acteur de mettre à jour son comportement à la réception d'un message `update`. Le nouveau comportement est alors une fonction reçue dans le message.

Dans tous les cas, le système met en place les indirections et le contrôle requis de manière à assurer l'atomicité de la mise à jour. Ces mécanismes ont une granularité du niveau de la méthode ou de la fonction. Ils requièrent généralement que la méthode ou la fonction ne change pas de type. Ces systèmes peuvent permettre de remplacer plusieurs méthodes ou fonctions en une seule opération atomique. En Erlang, la portée de l'opération est limitée à un unique module.

2. Parfois, une opération de remplacement est définie lorsque l'état d'un composant remplacé doit être préservé.
3. Certaines implantations de smalltalk vont plus loin en rendant `become` symétrique, `a become: b` échange `a` et `b`.

La sémantique précise de l'atomicité de l'opération de mise à jour dépend du mécanisme. Le principal point de variation concerne le comportement vis-à-vis des appels actifs au moment de la mise à jour. La question de la maîtrise de ces modifications afin d'assurer le comportement global du logiciel pendant et après les mises à jour est alors centrale et plusieurs critères de correction peuvent être envisagés [P114]. Dans les machines virtuelles Java, la spécification indique que l'exécution des appels actifs continue avec l'ancienne version, conduisant à une cohabitation des versions successives. Par ailleurs, des systèmes comme Erlang permettent d'expliciter des appels qui court-circuitent l'indirection, de sorte qu'il est possible d'exécuter les anciennes versions même après la mise à jour. Dans ces deux plates-formes, plusieurs versions d'une même fonction peuvent s'exécuter simultanément. À l'opposé certaines approches comme K42 [P10], Ginseng [P82] et Ksplice [P12] visent à garantir qu'à chaque instant une seule version est active. Alors que dans le premier cas une simple indirection peut suffire à mettre en œuvre le mécanisme, garantir qu'au plus une seule version est active nécessite des techniques plus sophistiquées pour assurer la bonne synchronisation [P53]. L'aspect sémantique a été étudié notamment dans [P114, P105, P21, P54]. Il convient enfin de noter que dans le cas où plusieurs versions peuvent s'exécuter simultanément la gestion de données partagées devient complexe. En effet, si les différents appels en cours de la fonction reposent sur des données partagées mais vues avec des types différents, il faut mettre en place des moyens pour maintenir une vue cohérente de ces données [P113].

Ces techniques ont en commun de contraindre les possibilités de mise à jour à la fois sur le moment de mise à jour (par exemple, une fonction ne peut être active), la façon dont est réalisée la mise à jour (par exemple, la fonction ne peut pas être appelée pendant la mise à jour) et les possibilités de mise à jour (par exemple, tous les sites d'appel de la fonction seront des sites d'appel de la nouvelle version [4]). D'autres techniques comme ReCaml (voir 5.3 page 83) et UpStare [P74] permettent de rendre au développeur une marge de manœuvre plus importante. Dans ces approches, le développeur peut choisir son critère de cohérence mais aussi le déroulement exact de la mise à jour. Cette liberté s'obtient cependant au prix d'une complexité d'utilisation élevée : l'algorithme de mise à jour doit être programmé à la main spécifiquement pour chaque logiciel, en manipulant directement l'état d'exécution.

Bilan Cette présentation rapide de certains des travaux importants sur le DSU a pour but de montrer que toutes les propositions actuelles sont le résultat d'un compromis :

- soit le mécanisme est simple, mais le comportement qui en résulte est grossier (Java, Erlang) ;
- soit la sémantique est saine, mais le mécanisme est rigide et les capacités de mise à jour réduites (K42, Ksplice et Ginseng) ;
- soit le mécanisme est souple, mais le coût est important (ReCaml et UpStare) ;
- soit l'impact de la mise à jour est clairement identifié, mais le paradigme de programmation est contraint et les capacités de mise à jour réduites (approche à base de composants).

Une partie de mes travaux récents dans le domaine consiste à essayer de fournir des mécanismes qui laissent le développeur libre de régler ce compromis selon ses besoins. Nous reviendrons en détail sur ce point dans la partie décrivant Pymoult 5.4 page 91. Cette section décrira d'ailleurs d'autres plates-formes.

4. Ce n'est pas le cas en Erlang.

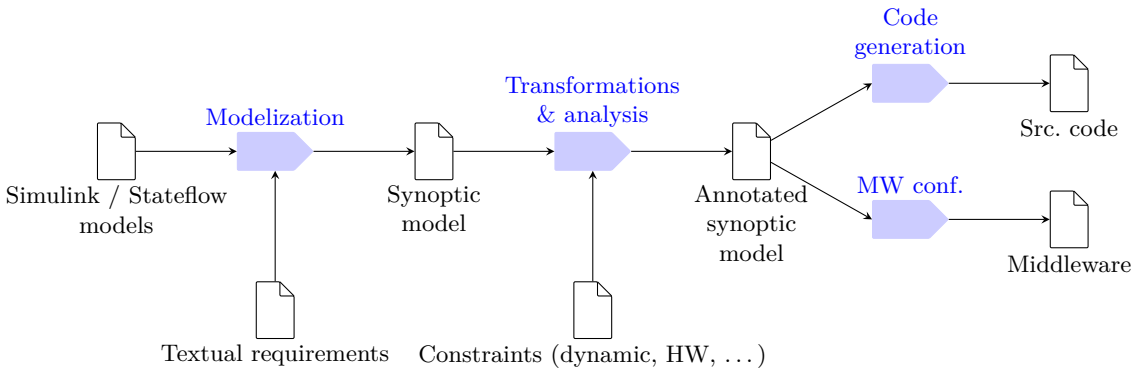


Figure 5.2 – Aperçu du processus de développement des logiciels de vol

5.2 Une méthode pour la mise à jour à chaud de satellites

5.2.1 Contexte

Un satellite est un engin spatial sans pilote dont l'architecture logicielle mais aussi matérielle est généralement spécialisée en fonction de sa mission. On distingue deux sous-systèmes : la charge utile qui représente les équipements applicatifs (par exemple l'instrumentation scientifique spécifique à la mission) et la plate-forme qui comprend la structure mécanique, les servitudes [5] utilisées par la charge utile et les communications avec les stations au sol. Nos travaux se sont focalisés sur le *logiciel de vol* (LV). Il s'agit du logiciel gérant la plate-forme et notamment en charge de l'avionique, du contrôle d'attitude et d'orbite, des interfaces de télémesure/télécommande, du contrôle thermique et énergétique, de la surveillance de l'état du système, et des services de bas niveau et de gestion du réseau embarqué (1553, SpaceWire, ...).

Une des principales contributions du projet SPaCIFY a été d'introduire les modèles au cœur des processus de conception, de développement et de maintenance des LV [16]. Ainsi, pour rendre accessible au domaine du spatial les technologies développées dans le cadre de l'ingénierie dirigée par les modèles, nous avons proposé un processus construit autour de modèles formalisés et outillés présenté dans ses grandes lignes figure 5.2. Les modèles sont exprimés en Synoptic [1], un langage de modélisation spécifique au domaine des LV de satellites défini par le projet. Synoptic permet de décrire des applications temps-réel composées de blocs fonctionnels et d'automates. Les modèles initiaux, résultant de l'activité des métiers du système et de l'automatique, sont enrichis par des propriétés non-fonctionnelles. Sur la base des contraintes matérielles à satisfaire, une architecture dynamique, décrivant par exemple la projection des blocs sur les *threads*, est dérivée. Enfin, le modèle résultant sert à générer le code source et la configuration de l'intergiciel du LV. À chaque étape, des analyses permettent de valider et de paramétrer les transformations. Les transformations de modèles formalisent l'expertise métier acquise par un industriel, elles lui sont donc propres.

Les logiciels de vols et leur conception doivent suivre un ensemble de contraintes [6] :

Temps réels lent Un LV est chargé de mettre en œuvre des lois de contrôle-commande. C'est donc un ensemble de tâches exécutées à des périodes fixes par un noyau temps-réel.

5. Ensemble des éléments qui permettent le fonctionnement d'un aéronef ou d'un navire (par exemple, l'électricité, le carburant, les systèmes de régulation thermique...).

6. Une description plus complète et détaillée peut être trouvée dans [P109, P110].

Chaque tâche réalise une servitude du système suivant de fortes contraintes temporelles héritées de sa conception par les automaticiens. Le nombre de tâches varie selon l'approche adoptée : Thalès Alenia Space favorise un grand nombre de tâches [7] alors que EADS Astrium regroupe les calculs pour réduire le nombre de tâches. Bien que les contraintes soient fortes, l'échelle de temps est relativement lente et la période d'activation des tâches varie entre 100 ms et 100 s.

Ressources matérielles à faible capacité Les ressources de calcul embarquées dans un satellite sont peu nombreuses. Ainsi, la mémoire vive allouée au LV est généralement inférieure à 10 Mo. L'image binaire est par ailleurs stockée dans une mémoire EEPROM d'environ 1 Mo à 2 Mo. Les processeurs ont une puissance de calcul qui varie entre 20 MIPS et 100 MIPS [8]. Par ailleurs, un satellite évolue dans un environnement hostile qui provoque un vieillissement rapide des équipements dû aux chocs, aux variations thermiques et aux rayonnements subis [9]. Les ressources, initialement faibles, sont de plus en plus contraintes au fur et à mesure de la vie du satellite.

Une industrie du logiciel particulière Les satellites scientifiques sont en général des exemplaires uniques. En effet, chaque mission scientifique a un objectif spécifique qui nécessite des équipements particuliers et donc une plate-forme toujours différentes. Pour les satellites de télécommunication, la construction se fait en (petite) série à partir d'une même conception. Néanmoins, les aléas de fabrication, du lancement puis de l'environnement spatial font que chaque satellite devient unique. Ainsi, chaque satellite a un logiciel de vol spécifique. Même lorsque des copies sont initialement installées sur plusieurs satellites, ces LV tendent à diverger pour s'adapter à l'état exact de chaque satellite.

Le test est difficile Il est difficile d'effectuer des tests réalistes du logiciel de vol. En effet, l'environnement physique spatial ne peut être que simulé de manière approximative. Par exemple, la gravité au sol ne peut être que compensée en suspendant les éléments du satellite à des ballons d'hélium. Par ailleurs, les vibrations lors de la phase de lancement peuvent affecter certains composants de manière totalement imprévisible. Ainsi, il est impossible de prédire le contexte réel de l'exécution du logiciel de vol après le lancement. Au mieux, des simulations peuvent donner des indications sur la correction du logiciel. Enfin, vu qu'il s'agit souvent de très petites séries, il n'existe que peu d'historique, ce qui rend encore plus délicat la définition de tests réalistes.

Enfin, une fois un satellite lancé, les seules interventions possibles doivent être réalisées à distance. Il faut donc pouvoir observer l'état du satellite (matériels et logiciels) et commander à distance le LV à la fois pour des manœuvres et pour des actions concernant la plate-forme (mises à jour logicielle par exemple). En effet :

- L'environnement physique spatial ne peut être que simulé de manière approximative. Il est donc nécessaire de pouvoir observer les éventuels défauts de calibrage et de corriger le logiciel en conséquence.
- Le satellite, et donc le logiciel, doit survivre autant que possible au vieillissement et aux dommages des équipements. Cela signifie qu'il faut pouvoir installer des contournements logiciels de manière à pouvoir poursuivre autant que possible la mission. Il faut également être en mesure de détecter les problèmes à corriger.

7. De l'ordre de 25 tâches actives sur un total de 50.
8. 20 MIPS pour les ERC32 et presque 100 MIPS pour les LEON2 et LEON3.
9. Par exemple, les ions lourds ont tendance à détruire les cellules des mémoires.

- Compte-tenu du coût d'un satellite, sa durée de vie effective doit être la plus grande possible, il faut donc pouvoir mettre à jour le logiciel pour de nouveaux objectifs. Remarquons que cela permet de retarder le développement du logiciel en fonction de besoins ultérieurs et de ne pas installer tous les logiciels au départ de la mission.
- Lorsqu'un obstacle est détecté sur la trajectoire du satellite, il est nécessaire de commander au logiciel de vol une manœuvre pour éviter l'obstacle.
- Lorsqu'un *bug* est détecté dans le logiciel, sa correction doit pouvoir être installée à distance, sans mettre en péril le satellite.

Cette observabilité / commandabilité du satellite doit utiliser le lien de communication entre les stations au sol et le satellite. La connectivité est uniquement assurée lorsque le satellite est à portée d'une antenne au sol et le débit est en général assez limité.

L'ensemble de ces explications montre qu'il est essentiel de garantir que le satellite est toujours en mesure d'être commandé depuis le sol. En plus de réaliser des vérifications concernant le logiciel, des mécanismes défensifs matériels et logiciels sont généralement ajoutés par exemple pour redémarrer les calculateurs en cas de problème.

5.2.2 Les pratiques à l'époque du projet

Les pratiques utilisées par Thalès Alenia Space pour reconfigurer le LV étaient les suivantes :

- Deux processus sont utilisés pour élaborer une nouvelle image du LV :
  - Pour une mise à jour simple, on modifie directement le binaire précédent. Cette pratique est surtout employée lorsqu'il s'agit de changer la valeur d'un paramètre ou pour supprimer des instructions tels que des appels ou des branchements conditionnels.
  - Pour des modifications plus complexes, un nouveau code source est produit puis compilé. Cette technique est utilisée lorsqu'il est nécessaire d'ajouter un composant ou de modifier des opérations.
- Pour limiter le besoin en bande passante, la plupart des opérations de mise à jour sont envoyées au satellite sous forme de *patch*. Le téléchargement intégral d'une nouvelle image du LV n'est réalisé qu'exceptionnellement. Ainsi :
  - Toute modification doit chercher à limiter la taille des différences entre binaires. Pour cela, des règles de codage métier sont imposées et les directives offertes par le compilateur et l'éditeur de liens sont exploitées.
  - Les différentes mises à jour et les images des logiciels successifs qui sont déployés à bord des différents satellites opérationnels sont stockées au sol.
- Pour rendre le processus plus fiable, les modifications ne sont pas faites directement sur l'image du logiciel de vol en EEPROM. Les mises à jour sont stockées et sont appliquées uniquement lors du démarrage du système, une fois l'image du logiciel de vol copiée en RAM. En cas de problème lors de la mise à jour ou avec le logiciel mis à jour, l'image de référence stockée en EEPROM est démarrée automatiquement. De temps en temps, les mises à jour sont stockées en EEPROM et deviennent permanentes, l'image de référence en EEPROM est ainsi mise à jour.

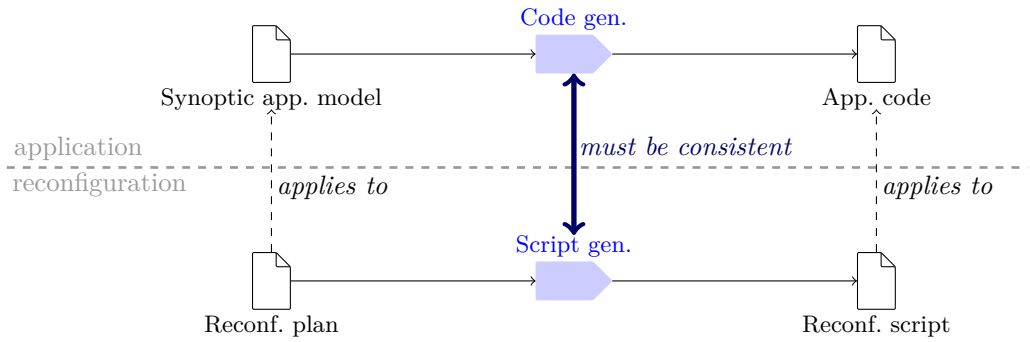


Figure 5.3 – Les générateurs de code pour l'application et la reconfiguration

5.2.3 La proposition d'ingénierie des reconfigurations

La proposition d'évolution faite dans la cadre du projet repose sur l'utilisation de l'ingénierie des modèles pour offrir une meilleur maîtrise du processus de mise à jour. Son principe suit celui de réalisation du LV également proposé dans le cadre du projet comme représenté en figure 5.3. Ce processus d'ingénierie des reconfigurations couvre les phases de conception, de développement et de validation et met en évidence deux formes de reconfigurations. À un haut niveau d'abstraction, les *plans* de reconfiguration portent sur le modèle Synoptic du logiciel et suivent alors le même processus que le LV pour être validés puis pour produire des scripts. De plus bas niveau, les scripts mettent en œuvre les reconfigurations en prenant en compte les détails d'implémentation du code (généré) du LV. Le générateur de code de l'application et celui des reconfigurations doivent être coordonnés.

De manière plus détaillé, le processus de conception d'une reconfiguration est présenté en figure 5.4 page suivante.

Ce processus commence par une phase de conception de la reconfiguration qui se base sur les exigences de la reconfiguration et sur les modèles Synoptic de la précédente version du logiciel. Pour cette phase, qui produit une première version du plan de reconfiguration, nous avons identifié trois démarches possibles :

- La conception est dirigée par le modèle du logiciel à mettre à jour (partie (a) de la figure 5.5 page suivante). Une activité de maintenance modifie le logiciel pour prendre en compte de nouveaux besoins et des rapports de défaillance. Le plan de reconfiguration est alors la liste des différences entre les deux versions du logiciel.

- La conception est dirigée par la reconfiguration (partie (b) de la figure 5.5 page suivante). À partir des besoins et des rapports de défaillance, le développeur réalise directement le plan de reconfiguration en se focalisant sur les changements qui doivent être appliqués au logiciel. Ensuite en simulant la reconfiguration, la nouvelle version du modèle du logiciel est produite. Ces reconfigurations peuvent éventuellement être appliquées à plusieurs variantes du logiciel.

- Il est également possible de suivre une approche hybride. Par exemple, il est possible d'identifer une zone du modèle que l'on souhaite reconfigurer par la première approche. L'intérêt de cette approche hybride est de pouvoir appliquer la reconfiguration obtenue à plusieurs logiciels à condition que leurs modèles contiennent le bout de modèle retenu. Attention, toutefois, chaque reconfiguration doit être revalidée systématiquement du fait des

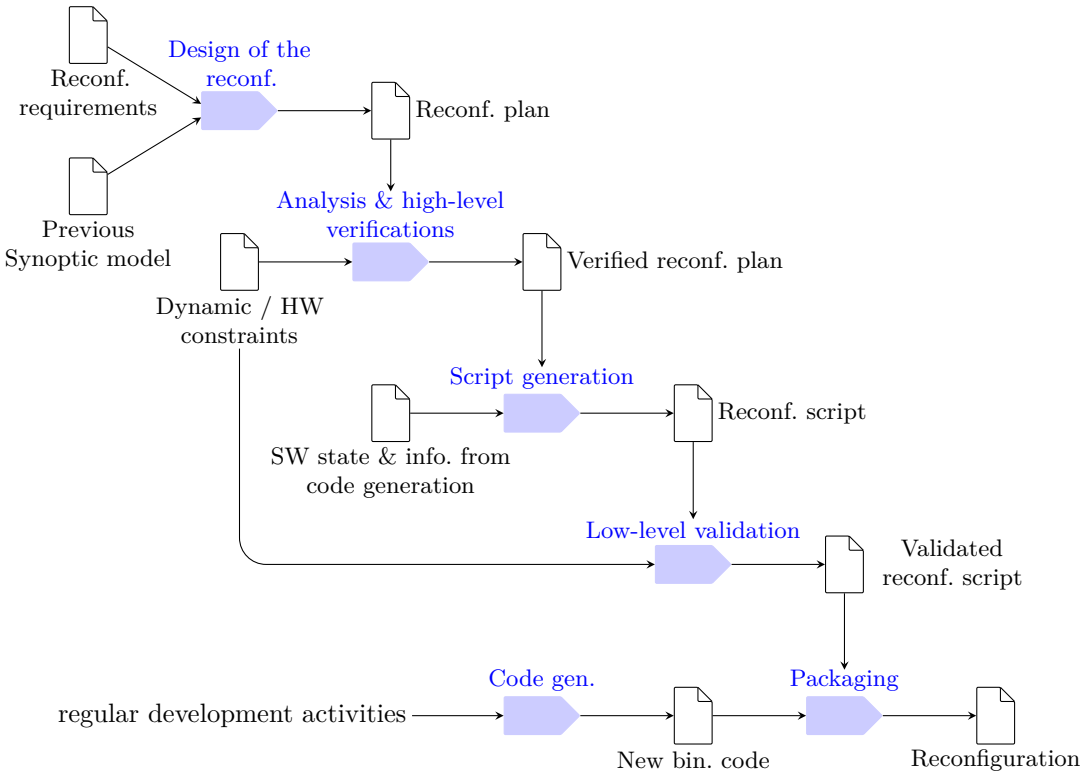
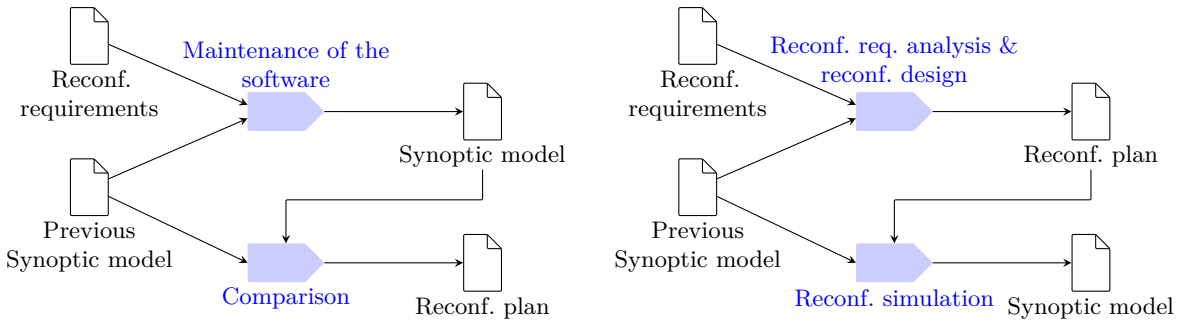


Figure 5.4 – Aperçu du processus de reconfiguration



(a) dirigée par le modèle du logiciel (b) dirigée par la reconfiguration

Figure 5.5 – La phase de conception d'une reconfiguration

contraintes globales des LV. Une fois validée, le plan de déploiement obtenu peut s'appliquer à tous les LV concernés.

Le processus contient ensuite des activités de vérification et de validation des reconfigurations dans lesquelles il va falloir s'assurer de pouvoir : (1) exécuter la reconfiguration dans l'état courant du satellite, (2) maintenir des propriétés spécifiques pour que le satellite puisse assurer un service minimum pendant l'exécution de la reconfiguration et (3) garantir que le résultat de la reconfiguration satisfait les nouvelles exigences. Parmi l'ensemble des propriétés à vérifier, on peut citer : des propriétés fonctionnelles, des contraintes spatiales et temporelles, des contraintes d'ordonnançabilité ou de qualité de service. Ces vérifications peuvent être réalisées soit au

niveau abstrait, soit après la phase de génération de code. La validation formelle d'un plan de reconfiguration peut être réalisée de manière globale, en prouvant la conformité du résultat de son exécution vis-à-vis des spécifications visées. Pour limiter l'étendue des vérifications à réaliser, nous proposons d'exploiter les éléments inchangés dans la spécification et dans le code. Il est possible de diminuer encore la quantité de preuve en utilisant un support de vérification modulaire, un système de contrats, par exemple [P48]. Ainsi, afin de déterminer l'étendue des vérifications à réaliser, un outil de configuration basé sur un système de contrat cherche à prouver pour chaque module (ou groupe de modules) à reconfigurer qu'il respecte le contrat initial. Si ce n'est pas le cas, il faut modifier son contrat. Alors, (1) les nouvelles garanties (plus faibles) du module ont un impact sur les hypothèses des modules qui l'utilisent, et / ou (2) les nouvelles hypothèses (plus fortes) du module ont un impact sur les modules qu'il utilise. Il convient alors de vérifier pour tous les modules impactés la compatibilité avec le nouveau contrat. Ce processus qui devait faire l'objet d'une suite au projet (abandonnée) n'a pas été complètement prototypé. Par contre depuis dans le cadre d'autre travaux, nous avons suivi une approche un peu différente qui consiste à prouver la reconfiguration dans Coq puis à utiliser son mécanisme d'extraction pour obtenir le script de reconfiguration vérifié. Cette approche est présentée en détail dans la section 5.5 page 104.

La dernière partie du processus proposé concerne la production du paquet de reconfiguration. Celui-ci contient le script de bas niveau mais aussi éventuellement de nouveaux composants logiciels. En effet, la reconfiguration peut aussi ajouter de nouveaux éléments dans le LV pour répondre à des besoins qui n'étaient pas identifiés lors de sa conception initiale. Le développement de ce nouveau code suit le processus standard d'ingénierie du logiciel (ligne du bas du processus).

5.3 ReCaml

5.3.1 Motivation

De façon à pouvoir valider certaines propositions du projet SPaCIFY tout en conservant un cadre plus simple que celui des langages synchrones et du C. Nous avons commencé à concevoir une version de Caml supportant la mise à jour à chaud. Dans ce cadre, nous avons poursuivi deux objectifs. Premièrement, nous souhaitions reposer sur un mécanisme offrant un contrôle total du processus de reconfiguration. De façon à pouvoir le rendre prédictible et capable de satisfaire à des contraintes temps-réel. Deuxièmement du fait des contraintes du domaine, nous souhaitions offrir des garanties et donc offrir des vérifications du code de reconfiguration. Notre choix s'est alors porté vers un langage fonctionnel pour ne pas avoir à gérer des effets de bord et sur la notion de type qui permet d'offrir des vérifications intéressantes à un coût supplémentaire relativement faible pour le développeur. Notons que le langage Caml était assez proche de Synoptic sur les aspects typage fort et décomposition fonctionnelle des programmes. Remarquons enfin que nous poursuivons actuellement des travaux sur la reconfiguration de programmes C et avons fait quelques expériences sur la reconfiguration de programmes synchrones.

5.3.2 Les principes de ReCaml

Lors de la conception de ReCaml, nous avons adopté plusieurs principes.

Tout d'abord, une reconfiguration est un programme à part entière. Dans ce programme, le développeur doit expliciter intégralement le comportement de la reconfiguration. En n'acceptant

aucun comportement implicite, la reconfiguration est plus facilement prédictible. Elle est en revanche plus complexe à produire, l'idée est alors de fournir ultérieurement des outils d'aide au développeur que ce soit sous la forme de méthodes, de patrons de réalisation voire de bibliothèques.

Ensuite, la reconfiguration doit être typée statiquement. Il s'agit d'une première étape pour rendre les reconfigurations plus fiables. Ce typage doit réutiliser le typage des programmes Caml classique et l'étendre pour toutes les opérations qui seront spécifiques à la reconfiguration.

L'idée est de modifier le moins possible le langage et de réutiliser des concepts ayant déjà fait leurs preuves. C'est ainsi que nous avons décidé de reposer sur la notion, maintenant assez répandue, de continuation et plus précisément celle de continuation délimitée [P40]. En ReCaml, l'état d'exécution d'un programme est capturé sous la forme d'une continuation délimitée. Notons qu'une telle continuation correspond à une pile d'activation de fonctions (des appels avec leurs paramètres). Ainsi, accéder à l'état de l'application consistera en le parcours de cette pile pour en extraire les données nécessaires.

En ReCaml, le principe de fonctionnement des reconfigurations est alors le suivant. Lorsqu'une reconfiguration doit être exécutée, le programme de reconfiguration est tout d'abord chargé en mémoire. Puis l'exécution de l'application est suspendue et son état est réifié sous la forme d'une continuation. Celle-ci est passée en paramètre au code de la reconfiguration, que nous appelons *compensation*. La compensation a pour rôle de faire le nécessaire, en fonction de l'état dans lequel l'exécution a été suspendue, afin d'intégrer la reconfiguration dans l'exécution. La compensation est donc une fonction qui prend en paramètre une continuation (l'état courant) et retourne une continuation (le nouvel état d'exécution). Pour construire la nouvelle continuation, la compensation peut employer tous les moyens à sa disposition, tels que finir des activités en cours ou exécuter des fonctions spécifiques. Lorsque la compensation estime sa tâche accomplie, la (nouvelle) continuation reprend alors son exécution. Notons que l'approche employée peut être considérée comme optimiste car l'interruption du programme pour la réalisation de la reconfiguration est immédiate. La charge de déterminer si la reconfiguration peut être réalisée ou non est alors assumée par la compensation. Si elle ne peut pas reconfigurer immédiatement, elle peut redémarrer l'ancien calcul après y avoir ajouté un point de relancement de la reconfiguration plus propice à la reconfiguration où celle-ci sera invoquée.

La compensation fonctionne donc en dépilant les appels de l'état capturé jusqu'à ce que soient traités tous les appels qui doivent l'être. Autrement dit, la compensation parcourt le graphe d'appel du logiciel à partir de l'état dans lequel la reconfiguration est déclenchée. L'environnement d'évaluation et ce qu'il *reste à faire* sont identifiés grâce au site de retour. Un type statique est associé à chaque site de retour afin de pouvoir vérifier que l'opération de décomposition / recomposition des continuations est utilisée correctement. Les traitements d'un appel peuvent consister à ne pas le toucher, à le supprimer, à le remplacer par d'autres appels ou même à le modifier. Ainsi, la compensation peut réaliser un grand nombre de politiques de reconfiguration. Elle peut, par exemple, continuer un calcul en cours, puis traiter le résultat avec de nouveaux calculs ou bien elle peut abandonner les anciens calculs et en recommencer de nouveaux. On obtient ainsi une liberté totale dans l'entrelacement entre l'ancien code et le nouveau code.

5.3.3 Un exemple d'utilisation de ReCaml

Pour illustrer ReCaml sans rentrer dans tous les détails, je vais reprendre l'exemple utilisé dans [14], le calcul des nombres de Fibonacci :

```
let rec fib n =
  if n < 2 then n
  else (fib (n-1)) + (fib (n-2))
in fib 12345
```

Cet exemple simple n'en demeure pas moins intéressant car il illustre les difficultés du DSU. En effet, la fonction `fib` est toujours active et est donc impossible à mettre à jour dans la plupart des approche de DSU existante. Nous verrons également que ses mises à jour ne sont pas complètement triviales à concevoir.

Nous allons discuter deux mises à jour :

- Tout d'abord, il convient de remarquer que la fonction `fib` va rapidement déborder. Il faudrait donc la mettre à jour pour utiliser des entiers de précision arbitraire. Sa première mise à jour vise donc à la transformer en la fonction suivante [10] :

  ```
  let rec fib_num n =
    if n < 2 then num_of_int n
    else (fib_num (n-1)) +/ (fib_num (n-2))
  ```

- La complexité de la fonction `fib` est très mauvaise ce que pourrait par exemple détecter un profilage de son exécution. On peut alors décider de la remplacer par une version avec mémoire qui va réduire sa complexité [11] :

  ```
  let rec fib' n i fi fi1 =
    if i=n then fi
    else fib' n (i+1) (fi +/ fi1) fi
  in let fib n =
    if n < 2 then num_of_int n
    else fib' n 2 1/ 1/
  ```

Remarquons immédiatement que la première mise à jour modifie le type de la fonction `fib`, il n'est donc pas possible d'utiliser une simple stratégie de *rebinding*. À l'inverse, la seconde peut être mélangée avec la première, ce qui permettrait d'utiliser une simple redéfinition de fonction. Par contre, cela ne permettrait pas d'atteindre la complexité de la version linéaire car la pile continuerait de contenir, au pire, n activations de l'ancienne fonction appelant toute la nouvelle version et conduisant ainsi à une complexité en n^2.

Une stratégie possible pour la première mise à jour consiste à ne conserver que les appels qui n'ont pas débordé et à convertir leurs résultats. Tous les appels qui ont conduit à un débordement sont abandonnés et remplacés par des appels à la nouvelle version.

Pour la seconde, une bonne stratégie consiste à chercher dans les résultats déjà calculés deux nombres de Fibonacci consécutifs et à utiliser la nouvelle version à partir de ces deux nombres. Si l'on ne trouve pas de telle paire, on recommence le calcul avec la nouvelle fonction de 0.

Avant de présenter la façon d'implémenter ces mises à jour en ReCaml, présentons l'approche que le développeur va pouvoir suivre. Pour réaliser une mise à jour, le programmeur doit pouvoir parcourir l'état réifié du programme : la continuation capturée. À chaque étape de ce parcours, il faut qu'il puisse déterminer à quelle étape du calcul on est et qu'il puisse extraire les données accessibles en ce point. Pour cela, nous allons utiliser les endroits de retour d'appel de fonction

10. En ReCaml, `num_of_int` est la fonction qui converti un entier classique en un entier à précision arbitraire ; `+/` est l'addition sur les entiers à précision arbitraire.
11. En ReCaml, `1/` est l'entier 1 en précisions arbitraire.

qui sont nommés par une étiquette [12]. Cette étiquette est ensuite utilisée lors du parcours de la continuation pour indiquer la position du calcul par rapport au code source. Ainsi, le code étiqueté serait :

```
let rec fib n =
  if n < 2 then n
  else (let fn1 = <L1> fib (n-1) in
        let fn2 = <L2> fib (n-2) in
        fn1 + fn2)
in
let _ = set_update_routine (fun r -> capture <Lupd> upto p as k in compensate r k) in
```

Il pourrait alors réaliser le parcours ci-dessous avec la fonction `match_fib_callers_` qui reçoit deux paramètres : `k` qui contient la continuation et `r` qui contient la valeur qui est en train d'être retournée (c'est déjà un `num`). Pour cela, on utilise l'opérateur `match_cont` qui voit une continuation comme une liste où chaque élément est une étiquette suivie d'un ensemble de valeurs. Ainsi, en `L1`, seul `n` est accessible et il reste à calculer `fib (n-2)`. On remplace alors l'appel à `fib` par un appel à `fib_num` et on l'ajoute à `r`. Ensuite, il faut revenir à l'appelant pour traiter l'appel antérieur, ce que l'on réalise récursivement. On procède de même en `L2` en s'assurant au préalable que `fn1` n'a pas débordé [13]. Enfin, une fois en `Lroot`, on peut reprendre l'exécution par la construction `reinstate`.

```
let rec match_fib_callers_ r k =
  match_cont k with
  | <L1:n>     :: tl -> match_fib_callers_ (r +/ (fib_num (n-2))) tl
  | <L2:n fn1> :: tl -> match_fib_callers_ ((ifnotover (n-1) fn1) +/ r) tl
  | <Lroot>    :: tl -> reinstate tl r
  | _  -> (* erreur *)
```

5.3.4 La démarche et langage

La démarche globale, présentée en figure 5.6 page ci-contre, a été :

- On part d'un λ-calcul standard avec appel par valeur et intégrant les définitions `let` et `let rec` de Caml. Il est présenté dans le bas de la figure sous le nom λ_{CBV}.
- On définit une extension de λ_{CBV} avec les opérateurs de manipulation de continuation à partir de la notion de continuation délimitée similaires à ceux de [P52] et [P36]. Le principe d'une continuation délimitée est de ne pas capturer tout le calcul en cours mais uniquement celui entre le point actuel et une marque précédente. Cela permet de mieux contrôler la partie de l'état courant qui est réifiée et d'éviter ainsi qu'il soit trop *gros*.

  Cette extension est représentée en partie gauche de la figure. Ses trois premiers opérateurs figurent dans les programmes. Le premier `prompt` x `in` e permet de positionner une marque fraîche de limite de capture `mark` m e, la variable x est alors remplacée par le nom de la marque m dans l'expression e. Le second, `capture` e_1 `as` x `in` e_2, évalue e_1 qui doit alors renvoyer un nom de marque m, la continuation est alors capturée jusqu'à cette marque et

12. Nous n'avons pas vraiment explorer la possibilité de générer automatiquement la version du code étiqueté mais cela semble intuitivement faisable.

13. On utilise la fonction `let ifnotover n r =if n >44 then fib_num n else num_of_int r`. En effet, si les entiers sont codés sur 32 bits, compte tenu du signe, le maximum est $2^{30} - 1$ qui est dépassé pour `n` valant 45.

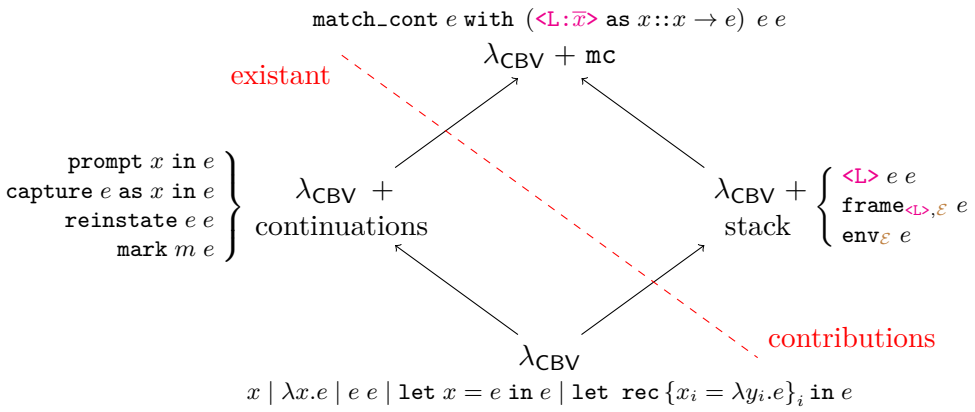


Figure 5.6 – La structure des modèles formels

remplace x dans e_2. Enfin, `reinstate` e_1 e_2 évalue ces deux arguments, puis redémarre le calcul de la continuation résultat de e_1 en lui donnant la valeur résultat de e_2.

– Il faut également donner une sémantique explicite à la pile d'exécution. Pour cela, on ajoute à λ_{CBV} l'étiquetage des appels de fonction. Il convient aussi de conserver la structure des blocs d'activation au sein du terme de façon à pouvoir les parcourir en cas de capture de continuation. C'est la construction $\texttt{frame}_{\texttt{<L>},\mathcal{E}}\ e$ qui indique que e s'exécute dans une activation créée par un appel en `<L>` et qui y retournera donc ensuite. L'environnement contient les valeurs qui sont dans l'activation (celles qui étaient accessibles lors de la création de l'activation). Enfin, la dernière construction $\texttt{env}_{\mathcal{E}}\ e$ indique que l'expression e est en train de s'évaluer dans l'environnement $\mathcal{E}$. La sémantique proposée pour modéliser le comportement de la pile suit de manière précise le comportement implanté dans le compilateur et la machine virtuelle de ReCaml.

– Enfin, il convient de définir l'opérateur qui permet de naviguer dans les continuations. La construction `match_cont` e_{cont} `with` (`<L:`$\overline{x}$`>` `as` x_{top}`::`$x_{rest} \rightarrow e_{ok}$) e_{no} $e_{\varnothing}$ va évaluer e_{cont} qui doit donner une continuation puis va l'analyser. Si la continuation est vide, le calcul se poursuit avec $e_{\varnothing}$. Si elle n'est pas vide, l'étiquette de son premier bloc d'activation (marqué par `frame`) est comparé à `<L>`. S'il ne correspond pas, on exécute e_{no}. S'il correspond, les valeurs de ses variables sont liées à $\overline{x}$, l'activation au sommet à x_{top} et le reste de la continuation à x_{rest} puis on exécute e_{ok}.

La sémantique opérationnelle complète du langage est présentée en détail dans [14]. Elle a été mécanisée en utilisant Coq.

Pour aider le programmeur de la reconfiguration à produire un code de reconfiguration correct, nous avons développé un système de type pour ReCaml. Ce système de type suit la discipline proposée par le λ-calcul simplement typé. Chaque type sera inféré et notre système est monomorphe. Les types possibles sont donc une variable de type noté α, un type de fonction noté $\tau \rightarrow \tau$, le type d'un *prompt* noté τ *prompt* ou le type d'une continuation noté $\tau \xrightarrow{\kappa} \tau$. Notons qu'en ReCaml contrairement à Caml nous ne supportons pas le polymorphisme, une variable de type représente donc un type inconnu et pas un type polymorphe. Le type du *prompt* correspond au type de la valeur qui va le traverser. Le type de la continuation mémorise le type du paramètre et celui du résultat.

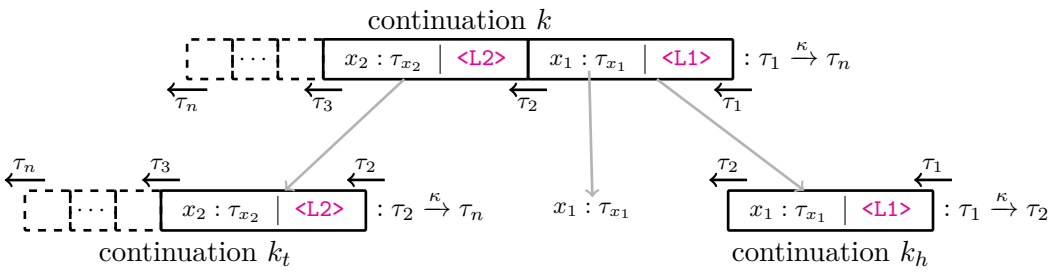


Figure 5.7 – Le principe de l'exécution et du typage d'une continuation.

Le typage suit des règles de la forme $\mathcal{E}, \mathcal{E}_{\mathcal{P}}, \mathcal{E}_{\mathcal{L}}, \tau \vdash e : \tau_e$ pour indiquer que l'expression e a le type τ_e dans les environnements de typage $\mathcal{E}$, $\mathcal{E}_{\mathcal{P}}$ et $\mathcal{E}_{\mathcal{L}}$. $\mathcal{E}$ conserve les hypothèses de typage des variables, $\mathcal{E}_{\mathcal{P}}$ celles des *prompts* et $\mathcal{E}_{\mathcal{L}}$ celles concernant les sites d'appel. Le type d'un site d'appel est un triplet $\langle \tau_{par}, \tau_{res}, \mathcal{E}_{\mathcal{V}} \rangle$ où τ_{par} est le type de la valeur reçue en retour de l'appel, τ_{res} est le type de retour de la fonction qui contient le site d'appel et $\mathcal{E}_{\mathcal{V}}$ est un environnement de typage qui stocke les types des variables accessibles. L'algorithme d'inférence calcule τ_e et $\mathcal{E}_{\mathcal{L}}$. La figure 5.7 résume le mécanisme du `match_cont` et son typage. La continuation k passée en paramètre, dont l'exécution doit se poursuivre au site de retour `<L1>`, est décomposée en deux sous-continuations k_h et k_t. L'appel en sommet de pile k_h, qui contient le site de retour `<L1>` est de type $\tau_1 \xrightarrow{\kappa} \tau_2$. Le reste de la continuation k_t, qui représente la suite du calcul de k_h est de type $\tau_2 \xrightarrow{\kappa} \tau_n$. Les valeurs contenues dans le sommet de pile peuvent être récupérée comme par exemple, ici, x_1. Pour typer ces valeurs et les sous-continuations, on utilise le type du site `<L1>` qui, ici, vaut $\langle \tau_1, \tau_2, \{x_1 : \tau_{x_1}\} \rangle$.

Il est important de bien comprendre qu'ici tous les types de la figure 5.7 sont statiques. Aucun calcul de type n'est réalisé à l'exécution ce qui permet de réaliser les vérifications dès la compilation de la reconfiguration.

Une machine virtuelle a été réalisée par extension de la ZAM2 [P72]. Celle-ci est une implantation fidèle de tous les mécanismes théorisés ci-dessus. Elle inclut donc un support pour la gestion des continuations. Pour cela, une continuation est réalisée sous la forme d'une tranche de la pile d'appel. L'instruction `prompt` est un pointeur dans la pile d'appel ; l'opération `capture` coupe le sommet d'appel jusqu'à ce pointeur pour en faire la continuation et enfin, `reinstate` remet la tranche de pile (la continuation) au-dessus du sommet de la pile d'appel avant de relancer l'exécution. Enfin, conformément à la sémantique opérationnelle, l'opérateur `match_cont` permet de parcourir une tranche de pile bloc d'activation par bloc d'activation. Pour cela, on utilise les informations sur la structure exacte de pile calculée par le système de type. Cette machine virtuelle réalise également les services de gestion de la mise à jour. Un programme est donné à cette VM avec toutes ses mises à jours, un fil d'exécution se charge alors d'écouter un port sur lequel il va recevoir des ordres de mise à jour. Il exécutera alors une fonction que le programme initial aura enregistré et qui sera en charge d'exécuter la mise à jour (opération `set_update_routine`). Par exemple, le code suivant réalise complètement la mise à jour de la section précédente.

```
prompt p ->
(* Version initiale *)
let rec fib n =
(* Nouvelle version *)
let rec fib_num n =
```

```
(* Mise à jour vers des entiers de précision arbitraire *)
(* Si n est plus grand que 44, r a débordé retourne fib_new n ou r converti *)
let ifnotover n r = if n > 44 then fib_num n else num_of_int r in
let rec match_fib_callers_ r k =
  match_cont k with
  | <L1:n>     :: tl -> match_fib_callers_ (r +/ (fib_num (n-2))) tl
  | <L2:n fn1> :: tl -> match_fib_callers_ ((ifnotover (n-1) fn1) +/ r) tl
  | <Lroot>    :: tl -> reinstate tl r
  | _  -> (* erreur *)
     (0/ -/ 1/)
in
let match_fib_callers r k =
  match_cont k with
  | <L1:n>    :: tl -> match_fib_callers_ ((ifnotover (n-1) r) +/ (fib_num (n-2))) tl
  | <L2:n fn1> :: tl ->
        match_fib_callers_ ((ifnotover (n-1) fn1) +/ (ifnotover (n-2) r)) tl
  | <Lroot>   :: tl -> reinstate tl (fib_num 12345)
  | _  -> (* erreur *)
     (0/ -/ 1/)
in
(* Compensation fib -> fib_num *)
let compensate r k = match_cont k with
    | <Lupd> :: tl -> match_fib_callers r tl
    | _  -> (* erreur *) (0/ -/ 2/)
in
(* Le programme principal avec l'enregistrement de la compensation *)
let _ = set_update_routine (fun r -> capture <Lupd> upto p as k in compensate r k) in
  <Lroot> fib 12345
```

Notons dans le code précédent que le principe d'une compensation correspond à un parcours du graphe d'appel car chaque site de retour désigne un arc de ce graphe. Le squelette algorithmique de la compensation d'une reconfiguration reposera donc naturellement sur des fonctions associées aux nœuds du graphe d'appel. La fonction associée à un nœud est mise en œuvre par une opération `match_cont`, qui contient autant de clauses que d'arcs arrivant à ce nœud. Chaque clause est responsable de compenser l'arc correspondant, réalisant une étape dans le parcours du graphe d'appel. À chaque étape, le développeur de la reconfiguration doit préciser ce qui doit être fait. Ainsi, par exemple, s'il faut poursuivre l'exécution de l'ancienne version, la sous-continuation qui capture l'appel au sommet de la pile peut être exécutée. Sinon, le développeur doit écrire le code correspondant au nouveau comportement souhaité. Pour cela, le développeur peut s'inspirer du code des différentes versions. ReCaml offre ainsi au développeur une maîtrise totale du comportement effectif de la reconfiguration. Remarquons que cela se fait au prix d'une complexité accrue. Nous considérons que cela est parfois le prix à payer pour pouvoir maîtriser la reconfiguration de systèmes très complexes sur lesquels pèsent de fortes contraintes de continuité de service.

Terminons cette section par une petite expérimentation menée dans le cadre du projet ReCaml. Nous avons simultanément au developpement de la VM pour ReCaml construit en Java une interface graphique de contrôle de la mise à jour. Cette interface permet de déclencher la mise à jour puis de suivre son exécution pas à pas à l'image d'un débogueur. Elle est construite de manière indépendante de la VM (appelé serveur ReCaml ci-après) et s'y connecte par des

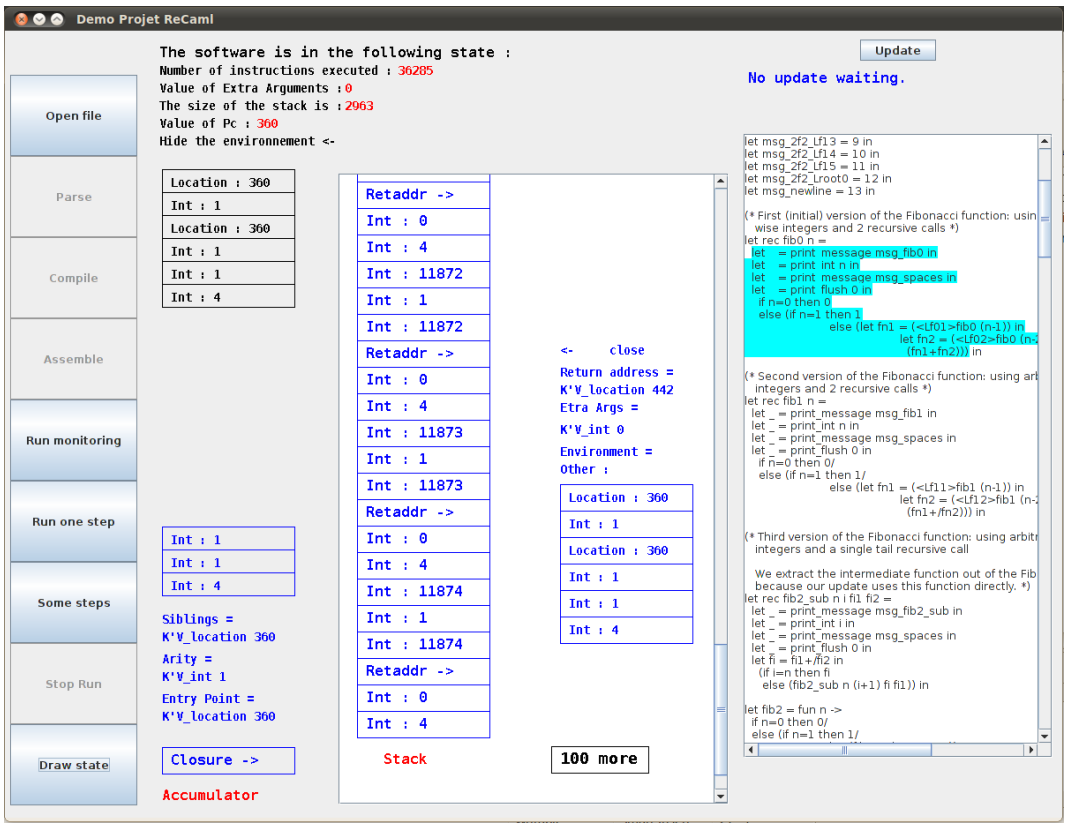


Figure 5.8 – Une interface de contrôle pour ReCaml

sockets de manière bidirectionnelle. L'interface présentée en figure 5.8 permet, suivant l'état du serveur ReCaml, de :

- ouvrir un fichier ReCaml, son code source est envoyé au serveur et est affiché dans la zone correspondante (lors de l'exécution la prochaine instruction à exécuter est surlignée en bleue) ;
- *parser*, compiler puis assembler le programme ReCaml afin de le rendre exécutable par le serveur ReCaml ;
- commencer ou reprendre l'exécution du programme ;
- commencer ou reprendre l'exécution du programme et avancer d'un pas (une instruction de VM) ou de quelques pas (jusqu'à un appel ou un retour de fonction) ;
- suspendre un programme en cours d'exécution ;
- afficher l'état du programme, le panneau central représente alors le contenu de la pile d'exécution et celui des registres, on peut explorer un peu plus en détail la pile en sélectionnant un bloc d'activation et on peut alors demander l'affichage de l'environnement courant ;
- déclencher une mise à jour.

5.4 Pymoult

Les travaux sur ReCaml et sur le DSU en général ont conduit directement aux constations suivantes :

- Il existe de nombreuses plate-formes qui reposent sur des hypothèses assez variées, adoptent des approches différentes pour satisfaire des besoins pas toujours identiques. Ainsi, certaines approches reposent sur l'explicitation de la structure du système logiciel qu'elles utilisent pour réaliser une mise à jour. D'autres approches requièrent des langages de programmation disposant de fonctionnalités spécifiques. D'autres encore s'accommodent des logiciels tels qu'ils existent, quitte à ne pas être en mesure de réaliser certaines mises à jour.

  En résumé, aucune des solutions proposées actuellement ne peut être appliquée de manière industrielle et massive du fait d'importantes contraintes soit sur la chaîne de production logiciel soit sur le type de logiciel que l'on peut produire.
- Les travaux existants dans le domaine du DSU montrent qu'il faut faire un compromis entre l'impact sur la chaîne de production des logiciels, les capacités de mise à jour et la complexité de leur mise en œuvre. Cependant, il n'existe pas aujourd'hui de technique permettant à l'ingénieur d'ajuster et de maîtriser ces réglages. Or, la définition d'une mise à jour doit être faite en fonction de la façon dont le logiciel a été construit. Cela impose au concepteur d'une mise à jour de connaître parfaitement la chaîne de production. Actuellement, ce n'est possible que dans le cadre de chaînes simples au fonctionnement uniforme (qui traitent tous les fichiers de la même façon). Par exemple, une chaîne de production qui utilise seulement un compilateur (`gcc` par exemple) pour compiler et lier tous les fichiers du logiciel est simple et fonctionne de manière uniforme.
- La réalisation concrète d'une mise à jour dépend fortement du logiciel mais aussi beaucoup de son état au moment de la reconfiguration. Une plate-forme qui offre des services de reconfiguration doit donc (1) offrir des services adaptables au besoin de la reconfiguration et (2) permettre au développeur d'une reconfiguration de choisir et configurer ces services conformément à ses besoins.
- Chaque plate-forme est spécifique à un domaine particulier (applications serveurs, systèmes d'exploitation, applications synchrones, ...) et leurs combinaisons est difficile. Cela pose un problème pour les applications complexes constituées de multiples composants aux exigences différentes. Ainsi, un logiciel réparti ou un logiciel réalisé en utilisant plusieurs langages de programmation différents ne sont pas reconfigurables avec les solutions existantes.

Sur la base de ces constatations, nous avons mené des travaux visant à proposer des mécanismes unificateurs avec des points de variation clairement définis. L'objectif étant de proposer un cadre général et une API générique pour le DSU.

Pour réaliser cette tâche, nous avons commencé par essayer de réaliser la plupart des mécanismes rencontrés dans la littérature dans une seule plate-forme : Pymoult. Pymoult est une part importante des travaux de thèse de Sébastien Martinez. Cette proposition que nous allons décrire ci-après, n'est pas encore complètement finalisée mais sa contribution me semble déjà notable et mérite donc d'être décrite ici.

Pour les travaux autour de Pymoult, nous avons fait le choix de travailler dans un environnement souple pour faciliter les travaux d'expérimentation. Nous avons choisi la plate-forme Python pour sa souplesse mais également pour sa popularité. Nous avons plus précisément choisi de travailler avec l'interpréteur Pypy [14] lui-même écrit en Python. Nos travaux ont d'ailleurs

14. Un portage sur CPython de Pymoult a depuis été réalisé également.

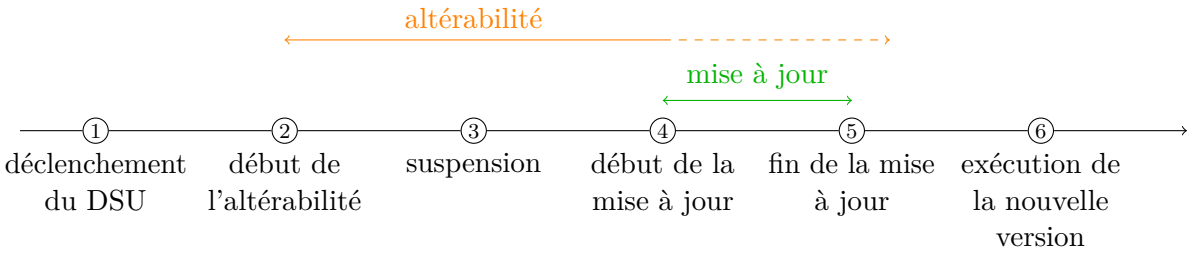


Figure 5.9 – Le cycle de vie de la mise à jour

mené à des extensions de cet interpréteur qui sont disponibles sous forme de *patchs*. Notons que nous avons adopté une approche ouverte, en choisissant, de travailler avec des logiciels libres et en produisant également des logiciels libres. Ainsi, toutes les réalisations sont disponibles publiquement sur bitbucket [15]. Les extensions à Pypy sont les suivantes :

1. une fonction peut être appelée à chaque création d'un objet, elle est positionnée par l'opération primitive `set_instance_hook(fun)` ;
2. les variables locales (qui sont donc dans la pile) peuvent être modifiées par la fonction `f_set_local(name,value)` qui a été ajoutée au type `frame` ;
3. il est possible de récupérer la liste des piles d'exécution de tous les threads sous la forme d'un dictionnaire dont les clés sont les identifiants des *threads* par `sys._current_true_frames()` ;
4. il est possible d'ajouter une fonction de trace (une fonction qui s'exécute après chaque instruction) à un *thread* quelconque [16] par `sys.settrace_for_thread(ident,fun,now)` (la trace est `fun`, le *thread* est `ident` et le dernier paramètre est un booléen qui permet s'il est vrai de mettre en place cette trace immédiatement).

Ces ajouts permettent alors de

1. maintenir une liste de tous les objets créés, ce qui fournit un moyen d'accès au tas, c'est utile pour les opérations de reconfiguration qui doivent convertir des objets ;
2. consulter et modifier la *vraie* pile d'appel et donc de changer les exécutions futures ;
3. permettre aux différents éléments de la reconfiguration de s'exécuter au milieu des calculs de l'application, ce que nous utilisons par exemple pour suspendre des *threads*.

5.4.1 Un cadre général pour le DSU

Afin de commencer à proposer une API générique pour le DSU, nous avons étudié en détail 17 [17] plates-formes de la littérature. Cela a permis d'identifier des problématiques générales que les plates-formes du DSU doivent traiter. Les plates-formes existantes répondent à ces enjeux en utilisant des mécanismes spécifiques assez variés. Ces variations peuvent s'inscrire dans un cadre commun.

15. https://bitbucket.org/smartinezgd/pymoult
16. En Pypy classique, ce n'est possible que pour le *thread* qui s'exécute.
17. Ces plates-formes ont été choisies pour leur diversité de programmes cibles et de modélisation du DSU : Kitsune [P56], Ksplice [P12], Opus [P8], K42 [P10], Ekiden [P55], Hotswap [P34], ReCaml [14], Polus [P28], Ginseng [P82], Javelus [P51], Rubah [P97], Jvolve [P107], ActiveContext [P113], Upstare [P74], DynamicML [P46], Proteos [P47] et Pymoult [28]. Une étude empirique systématique est en cours sur la littérature du domaine.

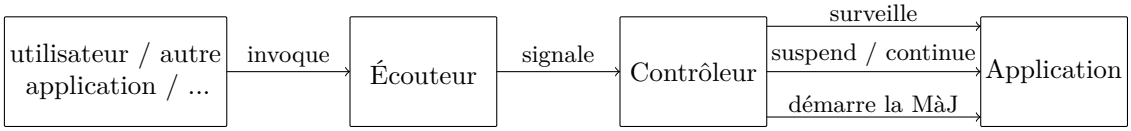


Figure 5.10 – Architecture fonctionnelle typique d'une plate-forme de DSU

Le cycle de vie Tout d'abord, on peut définir le cycle de vie représenté figure 5.9 page précédente.

1. La mise à jour est déclenchée soit par l'application, soit par une autre source comme une autre application la contrôlant par exemple.
2. L'application devient *altérable*, c'est-à-dire qu'elle est dans un état qui permet à la mise à jour de commencer.
3. L'application (ou une de ses parties) est éventuellement suspendue.
4. Puis la mise à jour commence à être appliquée.
5. La réalisation de la mise à jour termine. Si l'application était suspendue, son exécution peut reprendre.
6. La nouvelle version de l'application commence à s'exécuter.

Ce cycle de vie général doit bien sûr être adapté à la plate-forme et à la mise à jour. Par exemple, avec ReCaml, comme on travaille par reconstruction de la pile d'exécution, les mises à jour ne contiennent pas les étapes 5 et 6.

L'architecture L'architecture typique d'une application (voir figure 5.10) qui s'exécute dans une plate-forme de DSU repose en général sur les notions de :

- *écouteur*, composant en charge d'attendre les commandes de déclenchement des mises à jour ;
- *contrôleur*, composant qui est chargé de surveiller l'exécution de l'application à mettre à jour, de trouver le moment propice à la mise à jour, de suspendre l'exécution si c'est nécessaire, d'appliquer la mise à jour et de continuer l'exécution de l'application en fin de mise à jour.

L'altérabilité Le critère d'alterabilité est un point commun important des différentes plates-formes de DSU même si chacune d'elle en a sa propre interprétation. On trouve dans la littérature de nombreuses propositions de critères qui permettent de déterminer si une mise à jour est applicable à l'état courant de l'application. On peut citer par exemple la *quiescence* [P66], la *tranquillité* [P112] ou la *sérénité* [P45]. Le premier critère est le plus courant. Si l'on se place dans le cadre d'une mise à jour d'une fonction f, il correspond à trois conditions sur f : (1) f n'est pas en train d'appeler une autre fonction, (2) f n'est pas active et (3) il n'y aura plus d'appel à cette fonction f. Dans la pratique cela correspond à suspendre l'application et à s'assurer que la fonction n'est pas dans la pile d'appel.

5.4.2 Des choix variés et figés par plate-forme

Dans ce cadre commun, nous avons retenu dix-huit enjeux que nous avons classés en cinq categories. Ces enjeux nous semblent permettre à un ingénieur de faire un choix raisonné de plate-forme de DSU. Ils sont :

(A) la gestion de l'altérabilité

1 : le critère d'altérabilité est-il statique ou dynamique ?

2 : qui définit le critère d'altérabilité ?

3 : qui surveille l'état d'altérabilité en utilisant le critère ?

(B) la gestion des processus

4 : l'application (ou une de ses parties) est-elle suspendue pendant la mise à jour ?

5 : l'application (ou une de ses parties) est-elle redémarrée lors de la mise à jour ?

(C) le contrôleur

6 : peut-il être ajouté à la volée ?

7 : quelle est sa portée ?

8 : peut-il être spécifique à une (ou des) mise(s) à jour ?

9 : où est-il ?

10 : la mise à jour se fait elle à l'initiative de l'application ou du contrôleur ?

(D) l'accès aux données et leur conversion

11 : une donnée peut-elle exister en plusieurs versions simultanément ?

12 : quelle est la stratégie d'accès aux données ?

13 : quand sont-elles converties ?

(E) la réalisation et la forme de la mise à jour

14 : comment redéfinit-on une structure de données ?

15 : comment redéfinit-on une fonction ?

16 : quel est l'unité minimale de reconfiguration ?

17 : la mise à jour est-elle embarquée dans l'application ?

18 : la mise à jour est-elle incluse dans l'application après sa réalisation ?

Les dix-sept plateformes analysées sont détaillées suivant ces critères dans les tables 5.1 page suivante et 5.2 page 96. Cette analyse permet de :

- démontrer la variabilité des choix faits. Par exemple pour la responsabilité de la surveillance de l'altérabilité il y a quatre choix différents ou surtout pour la forme d'une unité minimale de reconfiguration, onze choix différents sont faits. La dernière ligne en fond jaune de chacune des tables 5.1 et 5.2 donne cette valeur, sa moyenne est de 4,3 approches possibles pour chacun des enjeux.
- la variabilité des choix fait par les plates-formes. Les dix-huit plates-formes font toutes une combinaison de choix différents. La tables 5.3 page 98 contient une mesure de corrélation de choix pour chaque plate-forme prise deux à deux, il s'agit du pourcentage de choix de mécanismes identiques. La moyenne est à 49 % pour une médiane à 50 %. Certaines plates-formes sont assez proches (plus 84 % de choix commun) mais elles ont toujours un choix qui change de manière assez importante leurs propriétés. Par exemple, Ekiden se distingue de Kitsune par le choix de l'initiative de la mise à jour [18]. Rubah a choisi une approche par rechargement de classe qui le distingue de Kitsune.

18. D'autres différences existent bien sûr dans l'implémentation ou la gestion des processus. Mais elles ne font pas partie des enjeux retenus pour cette analyse.

	Altérabilité			Gestion des processus		Contrôleur				
	évaluation	définition	surveillance	suspension	redémarrage	à la volée	portée	spécialisé	place	initiative
Kitsune	statique	app dev	application	oui	oui	non	appli	non	plate-forme	autre
Ksplice	dynamique	plate-forme	contrôleur	non	non	non	appli	non	plate-forme	autre
Opus	dynamique	plate-forme	contrôleur	oui	non	oui	appli	non	plate-forme	autre
K42	dynamique	plate-forme	contrôleur	en partie[1]	non	oui & non[2]	variable[3]	non	plate-forme	autre
Ekiden	statique	app dev	application	oui	oui	non	appli	non	plate-forme	appli
Hotswap	dynamique	plate-forme	VM	oui	non	non	appli	non	plate-forme	autre
ReCaml	statique	plate-forme	VM	oui	modif pile	non	non	non	plate-forme	autre
Polus	aucun			oui	non	non	appli	non	plate-forme	autre
Ginseng	statique	app dev	application	oui	non	non	appli	non	plate-forme	autre
Javelus	dynamique	plate-forme	VM	oui	non	oui & non[4]	variable[5]	non	plate-forme	autre
Rubah	statique	app dev	application	oui	oui	non	appli	non	plate-forme	autre
Jvolve	dynamique	plate-forme	VM	oui	non	non	appli	non	plate-forme	autre
ActiveContext	aucun			NS	non	non	appli	non	plate-forme	autre
UpStare	statique	plate-forme	contrôleur	oui	modif pile	non	appli	non	plate-forme	autre
DynamicML	dynamique	plate-forme	contrôleur	oui	non	non	appli	non	plate-forme	autre
Proteos	dynamique	mise à jour	plate-forme	en partie[6]	processus	non	appli	non	plate-forme	autre
Pymoult	choix	choix	contrôleur	choix	choix	choix	choix	choix	choix	choix
Nb de choix	3	4	4	5	5	5	5	2	2	3

TABLE 5.1 – Les trois premières catégories (Altérabilité, Gestion des processus et Contrôleur)

[1] uniquement les *threads* existants
[2] non pour les *threads*, oui pour les composants et mediateurs
[3] globale pour les *threads* et au niveau du composant pour les mediateurs
[4] non pour le contrôleur général et oui pour les adaptateurs de méthode
[5] globale sauf pour les adaptateurs qui portent sur une méthode
[6] uniquement les processus impactés

	accès aux données et conversion			**réalisation et forme de la mise à jour**				
	contexte	strategie d'accès	conversion	redef de structure	redef de fonction	unité de reconf	embarqué	dans l'appli
Kitsune	non	immédiat	à la demande	complète	complète	application	non	oui
Ksplice	non	aucun	aucun	NS	saut	fonction + donnée	non	module
Opus	non	aucun	aucun	NS	saut	fonction	non	librairie dyn.
K42	sorte[1]	immédiat	à la demande	complète	pointeur	composant	non	oui
Ekiden	non	immédiat	à la demande	complète	complète	application	non	oui
Hotswap	non	immédiat	à la demande	ajout de méthodes	pointeur	méthode	non	oui
ReCaml	non	choix	choix	aucun	aucun	*thread*	oui	oui
Polus	sorte[2]	immédiat	au besoin	complète	saut	fonction + donnée globale	non	oui
Ginseng	non	progressif	au besoin	complète	indirection	fonction + donnée	NS	oui
Javelus	non	variable[3]	variable[4]	NS	indirection	classe	non	oui
Rubah	non	immédiat	à la demande	complète	recharg. de classe	application	non	oui
Jvolve	non	immédiat	à la demande	réécriture du *bytecode*	pointeur	classe	non	oui
ActiveContext	oui[5]	aucun	aucun	NS	NS	classe + contexte	non	oui
UpStare	non	immédiat	à la demande	complète	complète	application	non	oui
DynamicML	non	immédiat	à la demande	complète	complète	classe	non	oui
Proteos	non	immédiat	à la demande	complète	complète	processus	non	oui
Pymoult	en cours	choix	choix	choix	choix	choix	choix	choix
Nb de choix	4	4	4	4	6	11	3	4

TABLE 5.2 – Les deux autres catégories (Accès aux données et Mise à jour)

[1] les *threads* de différentes générations sont gérés différemment
[2] exécutions simultanées avec synchronisation des états
[3] progressive pour les objets et immédiates pour les classes
[4] au besoin pour les objets et à la demande pour les classes
[5] un contexte par *thread* avec synchronisation des états

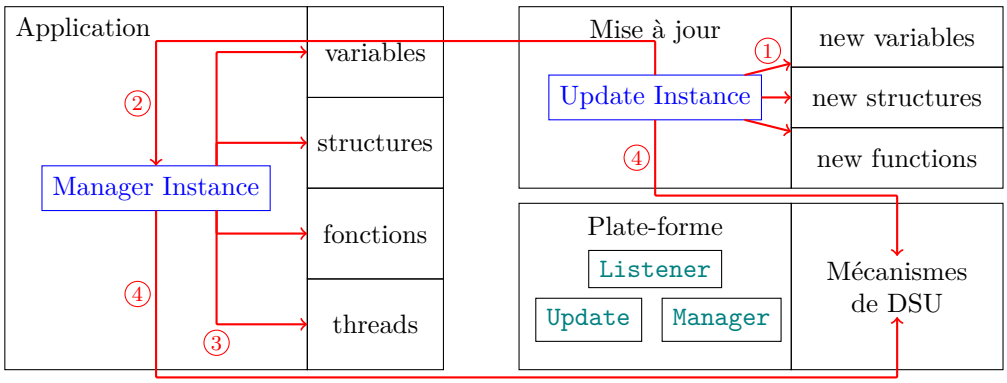


Figure 5.11 – Les classes `Manager` et `Update`

Cette analyse conforte Pymoult dans sa volonté d'assurer le maximum de souplesse en permettant de choisir son mécanisme. Ainsi, les dix-huit enjeux donnent lieu à seize choix possible de mécanismes. Le choix de la responsabilité de la surveillance de l'altérabilité est contraint pour clarifier les rôles et responsabilités des différents composants participant à la reconfiguration. Enfin, les contextes sont en cours de réalisation. Cette réalisation nécessite une extension assez importante de Pypy qui sera décrite rapidement dans les travaux en cours et futur en fin de ce chapitre.

5.4.3 Vers une API générique pour le DSU

L'API que nous proposons est composée de deux niveaux. Une couche haute qui propose des mécanismes préconfigurés et faciles à utiliser plus proches des services que les plates-formes usuelles de DSU offrent. Néanmoins pour permettre au développeur de la reconfiguration de garder un contrôle fin sur l'exécution de sa mise à jour, une couche basse est offerte. Celle-ci contient un ensemble de mécanismes de grain plus fin hautement configurables et composables. Pour le traitement des données, cette API considère la classe comme la structure principale. Ce choix n'est pas limitatif car l'API n'utilise aucune propriété spécifique de la notion de classe et donc convient tout à fait pour les structures de données des langages non objet [19].

Notre API suit l'architecture fonctionnelle déjà présentée et se base donc sur trois classes principales : `Manager`, `Update` et `Listener`. Les instances de la première classe sont en charge du contrôle des mises à jour. Celles de la seconde regroupent l'ensembles des tâches que la mise à jour réalise. Enfin, la troisième est en charge de la réception de la mise à jour et de son déclenchement. La figure 5.11 décrit les interactions et les rôles de ces classes et de leurs instances. Pour simplifier la représentation, nous fixons une hypothèse de présentation, les classes `Manager` et `Update` sont fournies par la plate-forme [20] (ici Pymoult). L'application est alors contrôlée par une instance de la classe qui accède aux éléments de cette application : ses variables, ses types, ses fonctions et ses fils d'exécution. La mise à jour fournit une instance de la classe `Update` (①) qui contient de nouvelles variables (ou de nouvelles valeurs pour des

19. Le niveau d'héritage utilisé est simple, ce qui permettrait assez facilement de le remplacer par un mécanisme de délégation dans un langage non objet.

20. Le développeur d'une mise à jour est bien sûr libre de s'organiser différemment. Il peut par exemple inclure le contrôleur dans son application et embarquer la classe `Update` dans sa mise à jour. Pymoult reste agnostique sur la structure effective utilisée pour définir ces classes.

	Kitsune	Ksplice	Opus	K42	Ekiden	HS	ReCaml	Polus	Ginseng	Javelus	Rubah	Jvolve	AC	UpStare	DML	Proteos	Pym
Kitsune	100																
Ksplice	42	100															
Opus	42	77	100														
K42	45	48	48	100													
Ekiden	95	36	36	39	100												
Hotswap	62	59	59	62	56	100											
ReCaml	45	36	36	28	39	50	100										
Polus	56	48	48	45	50	56	34	100									
Ginseng	71	50	44	36	65	53	48	59	100								
Javelus	42	48	53	48	36	65	48	42	50	100							
Rubah	95	42	42	45	89	62	45	56	71	42	100						
Jvolve	62	59	59	62	56	89	50	56	53	71	62	100					
ActiveContext	47	60	54	40	40	54	34	74	50	40	47	54	100				
UpStare	84	53	53	56	78	67	56	56	59	48	78	67	47	100			
DynamicML	73	65	65	67	67	78	45	62	59	65	67	84	54	84	100		
Proteos	67	48	42	50	62	62	34	50	48	42	62	62	47	67	73	100	
Pymoult	0	7	7	6	0	0	12	0	0	0	0	0	0	6	6	0	100

TABLE 5.3 – Pourcentage de choix identiques pour les critères par couple de plates-formes (moyenne à 49 % et médiane à 50 %)

variables existantes), de nouveaux types et de nouvelles fonctions. Cette instance commence par initialiser le processus de mise à jour et notifie (②) l'instance du contrôleur que la mise à jour est prête. Le contrôleur surveille alors l'application. Lorsque celle-ci atteint un état d'altérabilité qui convient le contrôleur applique les modifications de l'application spécifiées dans la mise à jour (③). Pour cela, elle travaille en synergie avec l'instance de la mise à jour et toutes deux utilisent les services de Pymoult (④).

Dans l'exemple ci-dessus, les classes `Manager` et `Update` sont fournies par Pymoult mais le développeur est libre de fournir ses propres classes et il contrôle également les moments de création de leurs instances. Cette repartition des rôles permet au développeur de combiner des mécanismes :

- prédéfinis (venant de Pymoult) ;
- définis dans le contrôleur qui peuvent ainsi avoir été définis en même temps que l'application ou alors pour un ensemble de mise à jour ;
- définis dans la mise à jour, ils peuvent alors être adaptés à la mise à jour.

Notons que la cardinalité des relations entre applications, contrôleurs et mises à jour est configurable. Certaines applications auront un unique contrôleur qui réalisera toutes les mises à jour, d'autres auront des contrôleurs par type de mise à jour voire pour chaque mise à jour.

Les mécanismes fournis par Pymoult peuvent être de deux niveaux. On présente tout d'abord, les mécanismes de bas niveau qui permettent de manipuler l'état d'exécution. Puis nous présentons comment ces mécanismes de bas niveau peuvent être assemblés pour construire les mises à jour.

Les mécanismes de bas niveau

(A) la définition de l'altérabilité

1 : `isFunctionInStack(fun,thread)` vérifie si la fonction `fun` est dans la pile du *thread* `thread` ;

2 : `isFunctionInAnyStack(fun)` vérifie si la fonction `fun` est dans la pile d'un des *threads*, pour cela, le contrôleur maintient une liste des *threads* de l'application ;

3 : `staticUpdatePoint(name=None)` permet au développeur d'indiquer explicitement dans son code que l'application a atteint un point d'altérabilité dont le nom peut être donné en paramètre ;

4 : `wait_static_points(threads)` est une fonction bloquante qui attend que tous les *threads* de la liste `threads` atteignent un point de mise à jour statique.

(B) la gestion des *threads*

5 : `resetThread(thread)` permet de redémarrer le *thread* `thread` ;

6 : `switchMain(thread,fun,args=[])` permet de changer la fonction principale d'un *thread* `thread` en la fonction `fun` en lui donnant éventuellement des arguments ;

7 : `suspendThread(thread)` suspend le *thread* `thread` ;

8 : `resumeThread(thread)` reprend l'exécution du *thread* `thread` ;

(C) l'accès aux données et leur conversion

9 : `setLazyUpdate(type,fun)` met en place un accès progressif aux objets de type `type` avec une conversion paresseuse utilisant la fonction `fun` ;

10 : `startEagerUpdate(type,fun)` démarre la conversion immédiate des objets de type `type` en utilisant la fonction `fun` ;

11 : `DataAccessor` est une classe qui fournit un parcours sur un ensemble d'objets suivant une stratégie, le type des objets et la stratégie sont fixés lors de son instanciation, deux stratégies sont possibles `"immediate"` ou `"progressive"` ; lorsque le parcours est immédiat c'est un itérateur classique sinon il se bloque en attente de nouveaux objets qui lui sont envoyés au fur et à mesure ;

12 : `ObjectPool` fournit un singleton qui maintient une référence (faible) vers tous les objets créés, il est par exemple utilisé dans le cas d'un accès immédiat ;

13 : `HeapWalker` définit une classe *abstraite* qui représente un ensemble de fonctions qui seront appliquées à des éléments, le développeur doit concrétiser cette classe pour fixer les opérations à réaliser ;

14 : `traverseHeap(walker,module_names=["__main__"])` est une fonction qui parcourt les éléments du tas du module[21] fournis en paramètre, elle leur applique le *walker* `walker` ;

(D) la modification des structures de données et des instances

15 : `redefineClass(mod,otype,ntype)` redéfinit la classe `otype` du module `mod` en lui donnant la valeur de `ntype` ;

16 : `addFieldToClass(type,name,val)` ajoute (ou remplace) l'attribut `name` de la classe `type` en lui donnant la valeur `val` ;

17 : `updateToClass(obj,type,fun=None)`, l'objet `obj` se voit transformer en instance de `type` en appliquant éventuellement `fun` si elle est fournie ;

18 : `generateMixinUser(type,*mixins)` créé une nouvelle classe à partir de `type` et d'une liste de *mixins* `mixins` ;

19 : `applyMixinToObject(obj,*mixins)` la classe de l'objet `obj` est remplacée par la fusion de sa classe d'origine avec les *mixins* donnés en paramètre ;

(E) la gestion des liaisons et indirections

20 : `redefineFunction(mod,ofun,nfun)` redéfinit la fonction `ofun` du module `mod` en lui donnant la valeur de `nfun` ;

21 : `ProxyManager` est une classe qui permet de construire un *proxy* sur un objet, l'objet cible du *proxy* peut être modifié ;

22 : `proxify(obj)` produit un *proxy* du type précédent vers `obj` ;

23 : `reprouteProxy(proxy,obj)` change l'objet cible du *proxy* `proxy` pour qu'elle devienne `obj` ;

24 : `redirectPointer(pointer,target)`[22], le pointeur `pointer` pointe maintenant vers `target` ;

API de haut niveau À partir de ces fonctionnalités de bas niveau, il est possible de construire des mécanismes de plus haut niveau, plus faciles à utiliser par le développeur. Ces fonctionnalités sont fournies à travers un ensemble de classes spécialisant nos trois classes fondamentales :

21. En Python, il n'y a pas un tas global mais un tas par module.
22. En cours d'implantation.

`Listener`, `Manager` et `Update`. Un diagramme de classe simplifié 5.12 page suivante [23] résume les principales classes et méthodes fournies par Pymoult. Le principe général comme indiqué en rouge est qu'un des écouteurs reçoit une demande de mise à jour ①. Celle-ci contient soit :

- des informations de configuration qui permettent d'instancier une classe de mise à jour existante comme par exemple `SafeRedefineUpdate` en lui donnant le nouveau code de la fonction ;
- une classe qui spécialise une des classes de mise à jour comme par exemple une classe `CustomUpdate` qui spécialise `ThreadRebootUpdate` et redéfinit les méthodes `alterability` et `over`.

À partir de cela, l'écouteur instancie une classe de mise à jour (qui hérite de `Update`) ②. L'écouteur doit ensuite déterminer quel contrôleur il utilise et enregistrer l'instance de mise à jour auprès du contrôleur choisi ③. Le contrôleur exécutera ainsi la mise à jour lorsque ce sera son tour.

Dans Pymoult, nous fournissons donc :

- Une classe `SocketListener` qui réalise la classe `Listener` sous la forme d'un *thread* qui ouvre une *socket* sur laquelle elle attend la mise à jour.
- Une classe `ThreadedManager` qui définit des contrôleurs qui s'exécutent dans leur propre *thread*. Ils ont une queue de mise à jour. Leur comportement consiste à attendre qu'il y ait un élément dans cette queue pour l'exécuter. Il a alors en charge la vérification de l'altérabilité de l'application. Lorsque l'application est altérable, il s'occupe de suspendre tous les éléments actifs [24], d'appliquer la mise à jour puis de redémarrer tous les éléments actifs.
- Une classe `DSUThread` qui fournit une notion de *thread* que l'on peut redémarrer (`reset`) et à qui l'on peut donner une nouvelle fonction à exécuter. Pour cela, il utilise la notion de continuation de Pypy (les *continulets*).
- Cinq types de mise à jour :
  - `SafeRedefineUpdate` pour la redéfinition sûre de fonction en assurant que la fonction n'est active dans aucun des *threads*,
  - `ThreadRebootUpdate` pour le redémarrage d'un *thread* en lui fournissant éventuellement une nouvelle fonction à exécuter,
  - `HeapTraversalUpdate` pour l'application d'une fonction de conversion qui sera appliquée lors d'un parcours du tas,
  - `EagerConversionUpdate` qui applique immédiatement une fonction de conversion à toutes les instances d'une classe,
  - `LazyConversionUpdate` qui met en place immédiatement des mécanismes de conversion paresseuse des instances d'une classe.

À partir des mécanismes de Pymoult, il est possible de reproduire quatorze des seize plates-formes présentées. Il manque la gestion des contextes pour offrir les services d'ActiveContext et des opérations d'écriture de *frames* pour reproduire l'ensemble des mécanismes de ReCaml.

23. Tous les éléments ne sont pas donnés pour gagner en lisibilité. Les éléments en marron sont abstraits, les méthodes en bleu sont des redéfinitions et les trois classes en bleu-vert sont des *thread*.

24. Ce comportement pourrait être affiné par exemple par un développeur de mise à jour. Il devrait alors redéfinir la méthode `run`.

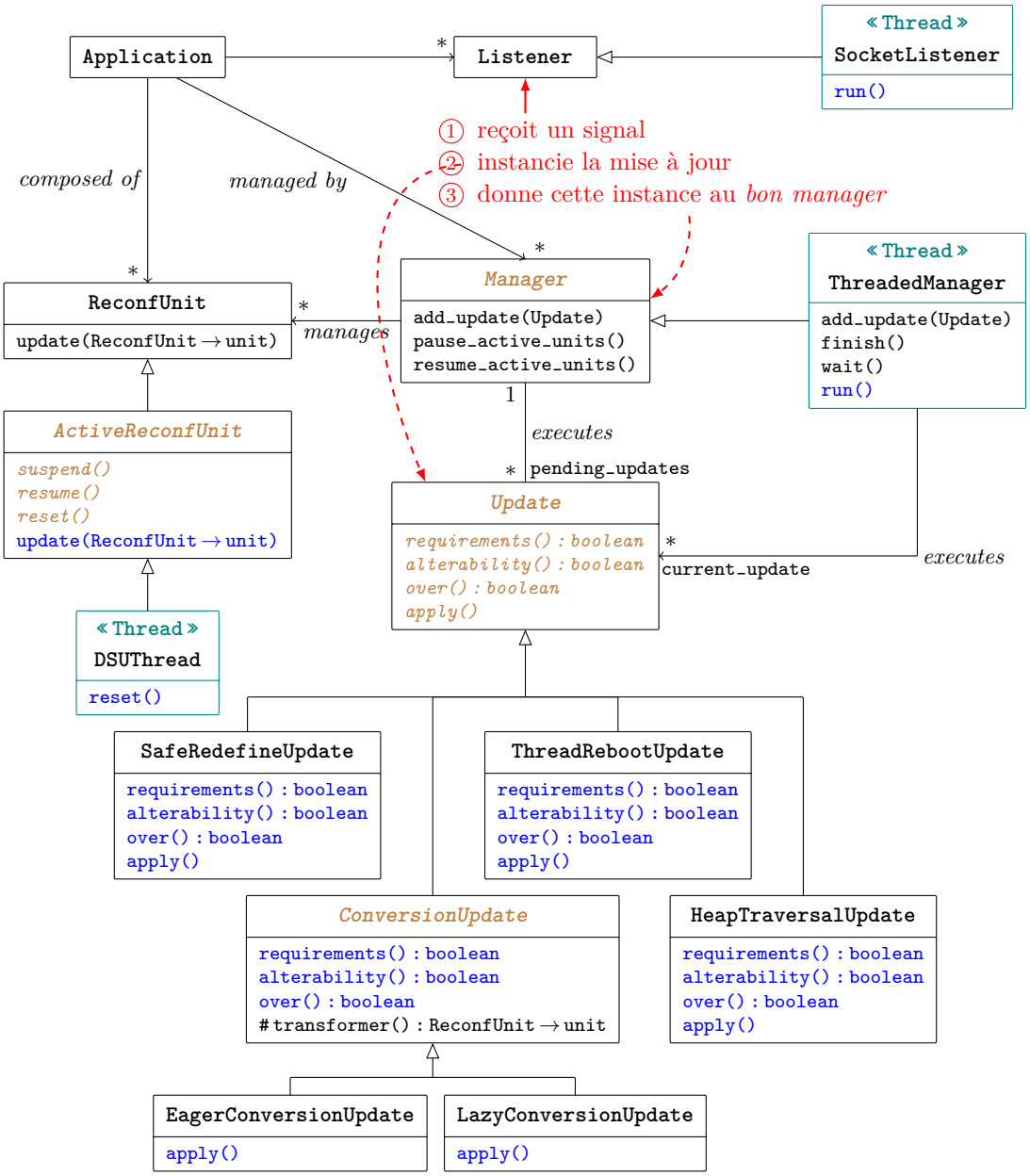


Figure 5.12 – Diagramme de classe de l'API haut niveau de Pymoult

5.4.4 Des expérimentations nombreuses

Pour conforter les propositions de Pymoult et leurs réalisations. Nous avons réalisé de nombreuses expérimentations d'usage de la plate-forme :

1. Nous avons appliqué Pymoult à la reconfiguration de programmes synchrones. Pour cela, nous avons étendu le compilateur du langage Heptagon [25] en lui ajoutant la possibilité de générer du code Python à la place du code C (traditionnel pour les langages synchrones). Ainsi, dans le cadre de son projet de master recherche, Rachid Ayoubi a pu expérimenter sur la mise à jour dynamique des programmes synchrones afin d'en identifier les problématiques propres. Il a exploré en détail deux points :
   - Une fonction synchrone est en général une fonction récursive dont l'état est codé par l'historique de cette récurrence. Par exemple, un système avec une réponse impulsionnelle sous forme d'échelon $G(p) = \frac{1}{p}$ est traduit par l'équation suivante $y_n = y_{n-1} + Tx_{n-1}$. Ce système dépend donc de son état précédent y_{n-1} et de son entrée précédente x_{n-1}. Si l'on souhaite ajouter un correcteur de type intégrateur : $F(p) = \frac{1}{p} * G(p) = \frac{1}{p^2}$. On aboutit à l'équation de récurrence $y_n = x_n + x_{n-1} + 2y_{n-1} - y_{n-2}$ qui introduit des dépendances vers les valeurs de l'instant $t - 2$.

     La solution adoptée repose sur un mécanisme de mise à jour en deux étapes. Premièrement, on ajoute un composant à l'application qui sauvegarde le nombre d'états nécessaires. Deuxièmement lorsque nous disposons de toutes les données la mise à jour est réellement faite.
   - Les signaux doivent être synchronisés pour être manipulés ensemble. Ainsi une équation $x_1 + x_2$ impose une relation entre l'horloge de x_1 et celle de x_2, il faut en effet que l'on puisse disposer simultanément de leur valeur pour réaliser la somme. Appliquer une reconfiguration qui change ces synchronisations est problématique et doit donc être empêchée directement sur le code Python. Le développeur doit réaliser la modification en heptagon et générer le nouveau code. La seule modification qui serait correcte serait de passer au nouveau code après avoir extrait les données nécessaires dans l'ancien.
2. Nous avons exploré la reconfiguration d'applications web. Pour cela, nous avons utilisé le serveur d'application Django [26] qui peut s'exécuter sur Pypy. Pour cela, des étudiants ont construit une application web assez simple qui a ensuite été mise à jour sans interrompre les sessions des utilisateurs. Ces travaux ont permis de travailler sur une unité de reconfiguration un peu particulière : la session. Ainsi, l'utilisateur connecté est informé de la mise à jour et peut choisir de continuer dans sa version ou demander la mise à jour de sa session. Cela a également permis de se confronter à un contexte plus délicat que les simples exemples réalisés précédemment. On a également pu tester, à travers la compréhension que les élèves en ont eu, la facilité d'utilisation de Pymoult.
3. Pour aller plus loin dans cette validation, Pymoult a été utilisé pour un cours master recherche sur la mise à jour à chaud dans le cadre d'un module sur les *Architectures pour Systèmes Informatiques Autonomes*. Cela a conduit à la construction d'un tutorial avec des exercices à réaliser par les étudiants [27]. Les exercices permettent de découvrir et comprendre une partie des différents mécanismes offerts par Pymoult. On part d'exemples simples à la *hello world* pour aller jusqu'à la mise à jour d'un serveur web simple.

25. Maintenant distribué à travers BZR http://bzr.inria.fr.
26. https://www.djangoproject.com
27. https://bitbucket.org/smartinezgd/pymoult/wiki/A%20simple%20Pymoult%20tutorial

4. Depuis un an, nous expérimentons le portage de l'API de Pymoult pour le langage C. Pour cela, nous sommes partis de la constatation que les actions de base employées lors de la mise-à-jour sont similaires à celles effectuées lors du débogage d'un programme. Ainsi, nous avons exploré cette piste pour construire des mécanismes s'appuyant sur les infrastructures de débogage existantes pour obtenir les deux propriétés suivantes : (1) aucune adaptation de l'application cible et (2) aucune modification de la chaîne de compilation. En effet, ces deux contraintes nous semblent indispensables pour passer du prototype de laboratoire à une application industrielle du DSU. Cette bibliothèque C, appelée Cmoult, contient déjà les mécanismes de mise à jour de bas niveau qui permettent d'explorer l'état de l'application (en utilisant le standard DWARF [28]), de modifier la valeur de variables globales et locales et de mettre à jour des fonctions. Nous travaillons actuellement au portage des notions de contrôleur et de mise à jour.
5. La dernière expérience en date est la mise à jour de Django lui-même. Il s'agissait ici de confronter Pymoult à une application réelle. Nous avons ainsi porté les passages de Django des versions 1.6.8 à 1.6.9 puis à 1.6.10. Ainsi, nous sommes capables de réaliser les mises à jour réelles d'un serveur Django sans interrompre son exécution.
6. Enfin, nous avons proposé un modèle de composant Pycots qui supporte la reconfiguration. Cette expérience un peu plus importante est décrite dans la section 5.5

5.5 Pycots et Coqcots

Coqcots est un modèle de composant défini formellement en Coq. Ce modèle de composant contient des mécanismes de reconfiguration qui permettent de modifier une application sans en stopper l'exécution. Ce modèle formel est utilisé pour garantir la correction d'un script de reconfiguration qui doit avoir été prouvé dans Coq. Ce modèle est implanté en Python par la plate-forme Pycots qui réalise cette infrastructure au moyen de la librairie Pymoult. Ces travaux ont fait l'objet d'une publication à CBSE [10]. Ils proposent les contributions suivantes :

- Un modèle de composant Pycots réalisé en Python et sa traduction en un modèle Coq. Des mécanismes bi-directionnels maintiennent en cohérence une application reposant sur Pycots et son modèle en Coqcots. Cette synchronisation est assurée dans un sens par un mécanisme d'introspection de Pycots qui permet d'extraire une description Coqcots d'une application en cours d'exécution. Et dans l'autre, par une extension du mécanisme d'extraction de Coq pour produire le code Python d'un script de reconfiguration.
- Une plate-forme d'exécution concrète de programmes suivant le modèle Pycots. Cette plate-forme utilise les mécanismes de DSU offerts par Pymoult pour supporter la reconfiguration.
- Un processus outillé pour supporter des reconfigurations architecturales prouvées d'applications Pycots. Cela inclut des aides à la conduite de la preuve qui offrent la possibilité de générer ses parties répétitives.

Pour présenter rapidement la démarche nous allons utiliser un exemple. Supposons une application dont une partie doit être reconfigurée. La figure 5.13 page suivante présente dans sa partie en haut à gauche cette architecture à base de composants. Elle contient six composants A, B et de C_1 à C_4. Chaque composant C_i utilise un service du composant A. Ces quatre dépendances permettent d'illustrer les différents cas que l'on rencontre lors de la reconfiguration

28. http://dwarfstd.org

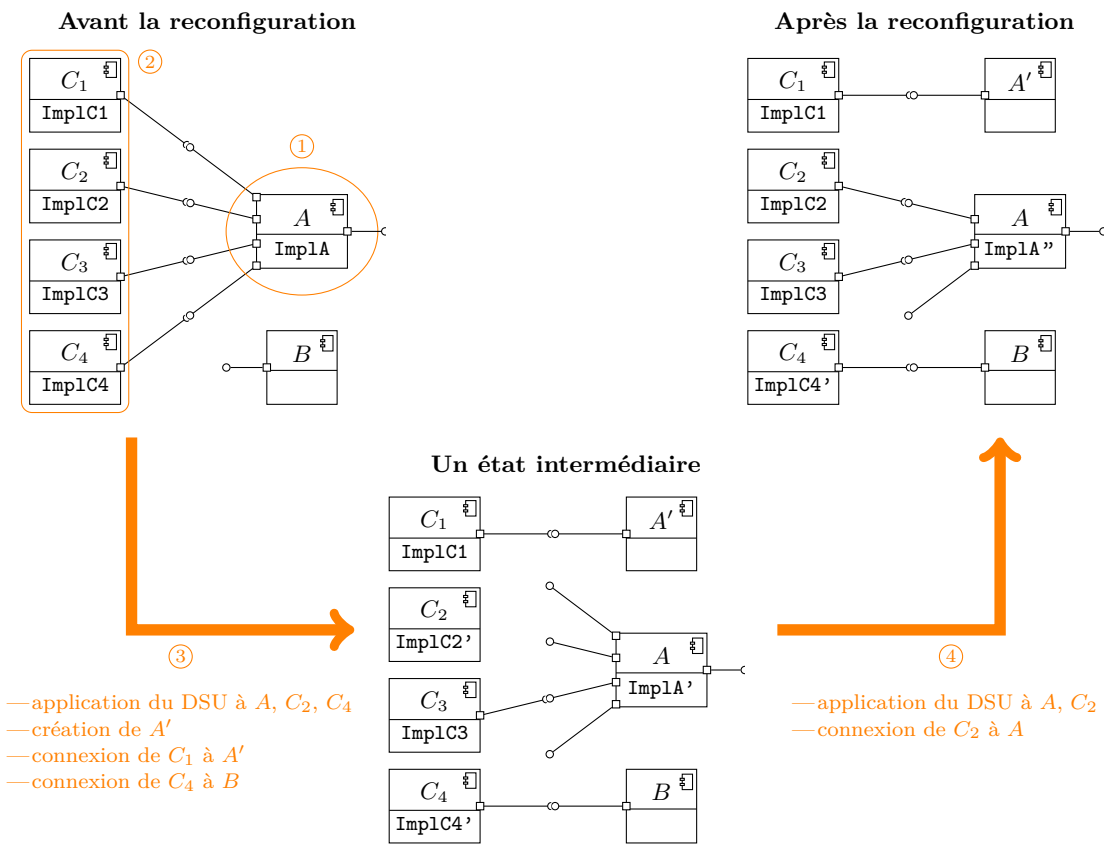


Figure 5.13 – L'architecture d'une partie d'application au cours d'une reconfiguration.

d'une application réelle. Ces composants peuvent être utilisés par d'autres composants qui sont ignorés ici car ils ne servent pas le discours. La seule contrainte imposée est qu'aucun autre composant n'utilise un service offert par A. Supposons que le composant A doit être reconfiguré pour retirer un des services qu'il fournit (le premier port utilisé par C_1 va être retiré). La première étape ① est de trouver les composants à reconfigurer. Ici, c'est trivialement A mais dans un cas réel cela peut être significativement plus complexe. Une fois les composants cibles trouvés, le concepteur de la reconfiguration doit identifier tous les composants qui utilisent un des services de l'un de ces composants. Ici, nous obtenons l'ensemble ② qui contient les composants de C_1 à C_4. Ensuite, il convient de décider comment on va cacher au reste de l'application le fait que le composant A ne sera qu'en partie disponible pendant sa reconfiguration. Dans l'exemple, nous avons choisi de :

1. conserver la même implantation au composant C_1 mais en le connectant à un nouveau composant A' à la place de A ;
2. modifier la réalisation de C_2 pour le rendre indépendant de A et ainsi permettre de le déconnecter de A ;
3. conserver C_3 ainsi que sa connexion à A.
4. mettre à jour le code de C_4 pour faire en sorte qu'il utilise le composant B à la place de A.

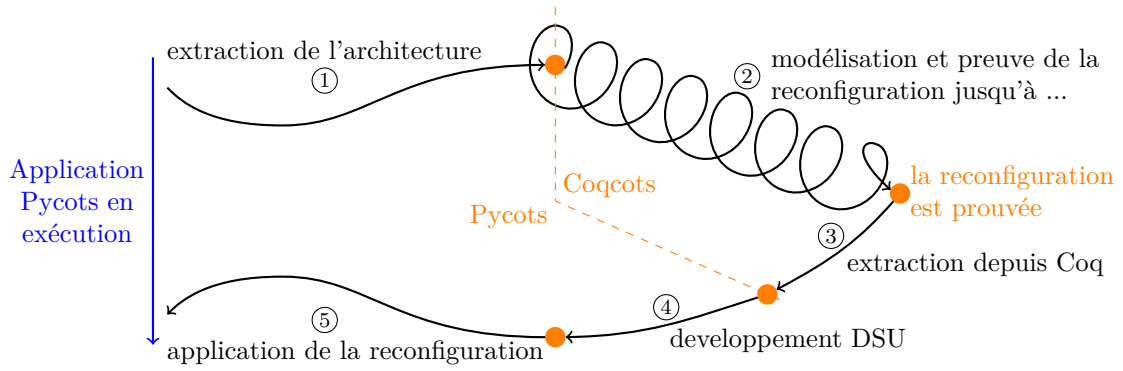


Figure 5.14 – Le processus de conception de reconfiguration sûre.

Dans une troisième étape ③, nous devons réaliser toutes les étapes conséquences de ces choix. Chaque modification d'une implémentation est faite en utilisant les mécanismes de DSU offert par Pymoult au sein de la plate-forme Pycots. Remarquons qu'il faut aussi mettre à jour A pour le préparer aux modifications prévues. Par exemple, il faut peut être suspendre certaines de ses *threads*. Dans cet exemple, A ne peut pas être complètement suspendu puisque C_3 continu de l'utiliser ce que doit prendre en compte sa réalisation `ImplA'`. Enfin ④, A peut être mis à jour comme prévu et les différentes mesures intermédiaires doivent être éventuellement annulées. Ici, par exemple, nous avons choisi :

1. C_1 va continuer à utiliser le composant A' ;
2. C_2 revient à son comportement initial et utilise A ;
3. C_3 est inchangé ;
4. C_4 va continuer à utiliser le composant B.

Cet exemple concrétise la notion de processus de reconfiguration et illustre le fait qu'un programme de reconfiguration est un programme assez complexe. Pour aider le concepteur d'un tel programme, nous proposons d'utiliser un assistant de preuve, ici Coq [29], pour concevoir une reconfiguration. Ainsi, nous favorisons la réalisation de reconfigurations correctes par construction.

Le processus que nous proposons est représenté en figure 5.14. La flèche en partie gauche représente l'application en cours d'exécution que nous devons reconfigurer. Lorsqu'une reconfiguration est requise, sa conception et son développement suivent les cinq étapes de la partie droite de la figure. Les étapes ② et ③ sont réalisées dans Coq et utilisent notre modèle abstrait de composant Coqcots. Les trois autres étapes manipulent des composants du modèle concret Pycots. Ces étapes sont :

① L'architecture actuelle de l'application est extraite en utilisant la couche réflexive de Pycots. Le résultat de cette extraction est un module Coq qui représente une architecture Coqcots correspondant à l'application. En Pycots, un composant est un objet (composant) qui encapsule un objet (contenu) masquant sa référence directe. Les services du composant sont réalisés par des méthodes de l'objet contenu mais exportées par l'objet composant sous forme de cloture. Le *framework* injecte les dépendances requises au sein de l'objet contenu qui possède alors un lien vers l'objet composant du fournisseur du service. Si le port n'est

29. https://coq.inria.fr

pas relié, ce lien est fait avec une méthode par défaut du *framework*. Ainsi, programmer un composant Pycots, c'est fournir la classe de son contenu (`Client` ou `Server` dans le code ci-dessous). Cette classe définit le code des ports fournis (`Server` définit une méthode `echo_port`). Enfin, il faut utiliser les primitives de la plate-forme pour créer les composants (`pycots.create`) et les connecter (`pycots.bind`).

```
import pycots
class Client(object):
    def __init__(self): super(object, self).__init__()
    def run(self): self.srv_port()
class Server(object):
    def __init__(self): super(object, self).__init__()
    def echo_port(self): print "hello"
c = pycots.create(Client, [], ["srv_port","optional_port"])
s = pycots.create(Server, ["echo_port"], [])
pycots.bind(c, "srv_port", s, "echo_port")
pycots.call(c, "run")
```

L'extraction de cette application mène à une architecture en Coqcots qui correspond. Un composant (client) a une implantation (i) qui utilise un ensemble de services (U, réduit à srv_port dans le cas du composant client) pour fournir un ensemble de services fournis (P qui est vide pour client). L'architecture contient alors les composants client et server et les liens entre le service fourni par server et utilisé par client (srv_port/echo_port). Dans Coqcots, un composant spécifie des préconditions sur l'architecture dans laquelle il sera instancié (cst pour le client). Cette contrainte permet par exemple de spécifier quel service est requis obligatoirement (Coq assure qu'il sera toujours lié). Pour cela, Coqcots fournit deux prédicats *contains* et *binds* pour exprimer qu'une architecture contient respectivement un composant ou une connexion.

```
Definition client_server := { a | ∃ client server,
  contains a client clt_use_facet clt_prov_facet clt_constraint (clt_implem a client)
  ∧ contains a server srv_use_facet srv_prov_facet srv_constraint (srv_implem a server)
  ∧ binds a client _ srv_port server _ echo_port }.
Definition client_constraint (self_arch: arch) (self: comp) := ∃ s provs port,
  binds self_arch self _ srv_port s provs port.
Definition server_constraint (self_arch: arch) (self: comp) :=True.
```

② Le concepteur travaille dans Coq pour définir et prouver sa reconfiguration.

Coqcots maintient un ensemble d'invariants sur une architecture correcte : (1) le typage correct des connexions, (2) l'existence des composants requis, (3) l'unicité du port fournisseur pour chaque connexion, (4) l'unicité de la définition de chacun des composants et (5) la satisfaction des préconditions de chacun des composants. Il a été prouvé par Jérémy Buisson que les opérations primitives de configuration de Coqcots préservent ces propriétés. Ainsi, en partant d'une architecture vide et en utilisant uniquement ces opérations, on aboutit forcément à une architecture correcte [30].

Les cinq opérations primitives que le développeur de la reconfiguration peut utiliser pour spécifier sa reconfiguration sont :

- *create* / *destroy* qui ajoute / retire un composant à / d'une architecture ;
- *link* / *unlink* qui connecte / déconnecte deux ports de deux composants d'une architecture ;

30. Cette preuve est elle-même faite en Coq et consiste en environ 2500 lignes de code Coq

– *hotswap* qui change le comportement d'un composant.

Il doit prouver que chacune des contraintes ci-dessus est maintenue durant sa reconfiguration et ce pour toute l'architecture. Pour cela, nous utilisons la technique classique dite du *frame axiom* [P57] pour garantir ces contraintes sur la partie non affectée par l'opération de reconfiguration par l'usage de postconditions sur ces opérations. Cette technique réduit le travail aux composants impactés mais rend nécessaire l'utilisation d'une tactique spécifiquement développée pour éviter à la preuve de voir sa taille exploser.

③ Une fois que la reconfiguration satisfait ses exigences et qu'elle est prouvée, le script de reconfiguration est extrait via le mécanisme d'extraction de Coq. Notons qu'il a fallu ici réaliser une extension de ce mécanisme pour qu'il génère du Python [31].

④ Le script extrait doit être relié au *framework* Pycots de façon à être relié aux objets Python qui s'exécutent. Pour cela, nous avons choisi de faire du script un foncteur Coq qui est donc un module prenant en paramètre l'ensemble des objets concrets sur lesquels il agit. Cette abstraction est réalisée sur la base de l'architecture qui a été extraite lors de l'étape ①. Le développeur doit donc écrire le code de glu pour relier le script aux objets représentant les composants Pycots. De plus, les différentes mises à jour nécessaires doivent être réalisées en utilisant les mécanismes de Pymoult.

⑤ Enfin, le programme Python obtenu est soumis à l'*écouteur* [32] qui le transmet au contrôleur Pycots. Celui-ci prend alors en charge la réalisation de ce programme de reconfiguration sur l'application.

Un exemple plus complet qui décrit un serveur web simplifié et sa mise à jour est décrit sur le site du projet : `http://coqcots.gforge.inria.fr`.

5.6 Bilan et perspectives

Ce chapitre résume mes principales activités dans le domaine de la mise à jour à chaud. Mes travaux ont donc concerné aussi bien les infrastructures et mécanismes de bas niveau nécessaires au DSU, que les abstractions à fournir au développeur, les outils de vérification de programmes de reconfiguration ou les processus d'ingénierie associés.

Certains travaux moins centraux ont été passés sous silence :

– Nous avons commencé à porter une partie des principes de ReCaml sur la JVM en utilisant une extension de la JVM pour disposer de continuations. Ces travaux sont complexes vu la taille effective et la complexité du code de la JVM. Pour l'instant rien de concret n'en est ressorti.

– Des travaux préliminaires sur la vérification formelle de la mise à jour en utilisant la théorie des catégories ont été conduits avec des collègues de l'ONERA [30]. Ils sont pour l'instant en suspens et non reliés au reste.

– Des travaux sur les langages de contrôle. Des expériences ont été menées sur l'utilisation du *scripting* dans la JVM mais aussi de combiner son utilisation avec OSGi.

31. Coq procède en deux étapes, une extraction vers un miniML puis la génération soit de Ocaml, Scheme ou Haskell. Il a donc fallu définir une traduction de miniML vers Python. En particulier, la définition de types inductifs et le filtrage de motif (*pattern matching*).

32. Au sens de ce qui a été présenté sur Pymoult.

– Des premiers travaux sur un langage fonctionnel de déploiement et contrôle d'application appelé *backstage*. Ces premiers travaux ont conduit à la soumission d'un projet ANR qui a été plutôt bien noté mais rejeté. Nous envisageons de le soumettre à nouveau dans le futur.

En terme de bilan, on peut mettre en avant la jeunesse du domaine et donc les nombreux travaux qui sont encore nécessaires pour disposer réellement de plates-formes utilisables à large échelle. Je pense avoir contribué notablement à l'avancée du domaine.

Les travaux en cours portent sur :

– La production d'un état de l'art empiriquement solide listant tous les mécanismes de DSU déjà publiés et les décrivant avec leurs intérêts, leurs usages et leurs limitations.
– Basé sur cette étude ainsi que sur des expérimentations de ré-implantation de plates-formes existantes, nous travaillons à rendre l'API de Pymoult plus complète. Dans ce cadre, nous travaillons depuis quelques mois à l'ajout de la notion de contexte dans Pypy.

  Un contexte rajoute un intermédiaire entre le programme et la mémoire. Ainsi, chaque nom utilisé dans le programme n'est plus relié directement à une valeur en mémoire (tas, pile, ...). Un nom est relié à une clé que le contexte relie à une valeur. Ce mécanisme d'indirection permet de modifier facilement les valeurs auxquels les noms sont reliés sans intervenir sur le programme. Il suffit, en effet, de modifier le contexte. De plus, en général, un contexte n'est valable que pour une partie du programme. Cela permet aisément de disposer d'un programme qui utilise des valeurs différentes suivant le contexte. Ainsi, si une tâche T_1 (resp. T_2) s'exécute dans un contexte C_1 (resp. C_2). L'exécution de la fonction `f` par T_1 conduit à l'exécution du code obtenu par $C_1(\mathsf{keyof}(\mathtt{f}))$ alors que pour T_2, on utilise $C_2(\mathsf{keyof}(\mathtt{f}))$. Le programme s'exécute donc dans deux versions simultanément.
– Des expériences sont conduites sur le portage des concepts de Pymoult en C. Une partie des mécanismes de bas niveau est déjà réalisée et repose sur les informations de débogage.
– Nous avons commencé à définir une sémantique formelle de la mise à jour à chaud dans le cadre d'une extension du λ-calcul. Une fois complet, ce calcul aura vocation à être utilisé pour formaliser les différents mécanismes. Il sera ainsi possible de mieux étudier les combinaisons de mécanismes et leurs propriétés.

Les perspectives que j'entrevois à l'issue de ces travaux sont les suivantes :

– À plus long terme, des travaux semblent nécessaires pour construire une chaîne de compilation et une méthode d'ingénierie permettant la mise à jour dynamique de ces applications complexes en utilisant tous les mécanismes existants. Cela implique de définir des procédés utilisant la sémantique formelle des mécanismes capables d'identifier les mécanismes utilisables pour chaque composant de l'application.
– L'établissement d'une API générique permettant la composition de mécanismes de DSU est nécessaire pour aller au-delà des limitations actuelles des différentes plates-formes existantes. Cette API générique permettrait d'inverser l'approche actuelle et donc de ne plus adapter une application à une plate-forme de DSU mais d'adapter une plate-forme pour une application. La conception et le développement des mises à jour nécessiteraient toujours une certaine expertise technique, mais elle serait plus abordable pour des applications très spécifiques (par exemple, les systèmes de contrôle de satellites) dont l'adaptation aux plates-formes de DSU disponibles est impossible.
– Rendre la couche basse de cette API neutre en terme de langage ou de système d'exécution sous-jacent est également un objectif important pour diffuser l'usage du DSU. Cela permettrait principalement deux choses :

  - la mise à jour de logiciels qui reposent sur l'utilisation de plusieurs langages, il est en effet devenu standard d'avoir des composants écrits dans différents langages ;
  - la séparation de la couche de bas niveau de l'API de sa couche de haut niveau, ce qui ouvrirait la porte à la conception de programmes de reconfiguration dans un langage différent de celui de la couche basse, par exemple, un programme Pymoult pourrait mettre à jour un programme écrit en C.

  Ainsi, il serait possible de commencer à songer sérieusement à la reconfiguration d'applications distribuées hétérogènes.

- Enfin, des études d'usage restent nécessaires. En effet, du fait de leurs contraintes, les plates-formes de DSU n'ont pas réellement été beaucoup utilisées dans des cadres pratiques et par des ingénieurs différents de ceux qui les ont conçues. Or, concevoir leur interaction avec des développeurs et les méthodes associées requiert cette étude. Quels seraient alors les usages du DSU ? On peut sans doute penser que les mises à jour de sécurité seraient les grandes bénéficiaires du DSU. En effet, leur application ne signifiant plus indisponibilité, leur application serait sans doute plus rapide et systématique. On peut néanmoins se poser les questions du risque de sécurité qui en découlerait et des mesures qu'il faudrait prendre pour éviter qu'une personne mal intentionnée ne mette à jour, à votre insu, une de vos applications pour y intégrer du code malveillant.

Chapitre 6

Des processus comme architecture d'activité

Ce chapitre synthétise des travaux de recherche menés depuis 2009 sur la modélisation en générale et plus particulièrement sur celle des processus. Dans le cadre, de ces travaux, j'ai dirigé la thèse de Fahad Golra [P49]. Ces travaux ont été publiés dans diverses conférences [11, 29, 12, 13]. Les principales contributions de cette activité de recherche sont :

- la proposition d'une approche multi-méta-modèle pour la définition de processus, chacun méta-modèle peut ainsi se focaliser sur une préoccupation ce qui améliore la réutilisation ;
- l'instanciation précise de cette proposition sur un modèle en quatre phases : spécification, implémentation, instanciation et exécution ;
- des méta-modèles multi-niveau qui permettent de séparer d'une part les abstractions contractuelles des réalisations, cela introduit de la souplesse qui permet d'offrir de l'adaptation à chaud des modèles de processus ;
- la réalisation d'un prototype comportant les différents outils nécessaires à la démonstration de notre proposition, l'utilisation de ces outils dans des expériences de laboratoire.

6.1 Motivation

Fuggetta [P42] définit un processus logiciel comme un ensemble cohérent de politiques, d'organisations, de choix technologiques, de procédures et d'artefacts qui sont nécessaires pour concevoir, développer, déployer et maintenir un logiciel. Selon son degré de maturité, une entreprise ou une organisation définit de manière plus ou moins détaillée ses différents processus logiciels. À l'image des systèmes logiciels, ces processus logiciels suivent un cycle de vie. À chaque étape du développement d'un processus logiciel, des préoccupations différentes doivent être abordées. Comme affirmé par Osterweil [P92], un processus a une nature et une complexité proche de celle d'un système logiciel. Il doit donc être traité de manière similaire. Ainsi, un processus logiciel doit être conçu (pour produire des spécifications), développé (pour décrire la réalisation des activités), déployé (pour être instancié à un projet et une équipe) et maintenu (pour surveiller sa mise en œuvre). Au cours du cycle de vie de développement d'un processus logiciel, l'ensemble des questions liées au processus sur lesquelles l'accent est mis change. Par exemple, durant la phase de spécification l'accent est mis sur la définition des artefacts qui seront traités ou développés au cours des différentes activités. Pour l'exécution d'un processus,

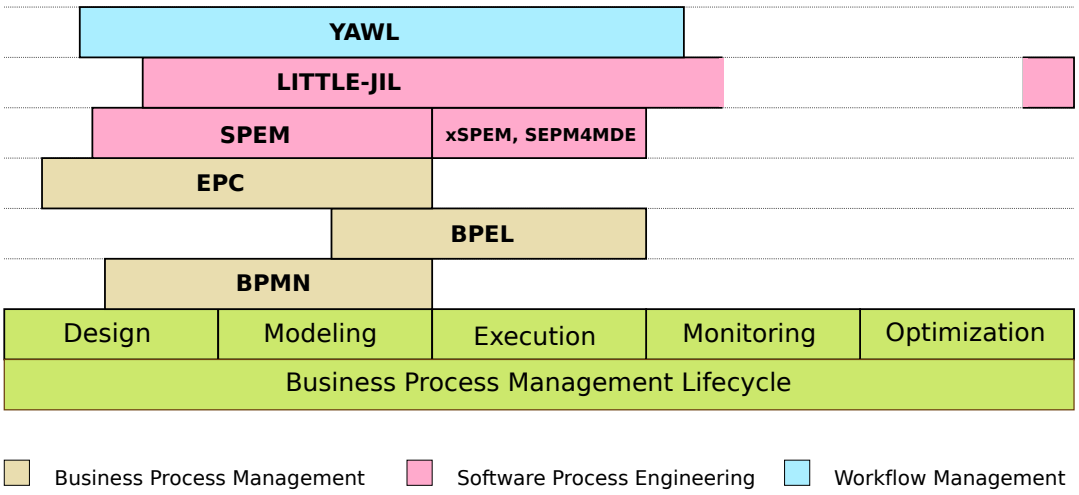


Figure 6.1 – La couverture du cycle de vie des processus

le focus se déplace sur les échéances temporelles qu'un artefact doit respecter. Enfin, pendant la phase d'exécution, la mise au point de l'activité ne correspond plus au choix de ses entrées et sorties, il est plutôt lié au « comment » et au « quand ».

Les modèles sont utilisés dans tous les domaines pour différentes raisons : prédire les propriétés d'un système, raisonner sur son comportement, communiquer entre les différents espaces techniques, etc. Dans le domaine de la production de logiciels, ils sont maintenant utilisés pour industrialiser le développement des systèmes logiciels [P64] par exemple en permettant l'automatisation de certaines tâches. Selon l'avancement du développement, différents aspects d'un système peuvent être considérés comme pertinents. De plus, chacun de ces aspects peut être modélisé en utilisant des langages différents pour des objectifs différents, offrant ainsi plusieurs perspectives sur un système étudié. Dans ce cas, un ensemble de règles peut être appliqué à des modèles d'une perspective pour les transformer en des modèles d'une autre perspective. Des règles peuvent également être définies pour enrichir des modèles en leur ajoutant des informations afin de les raffiner. Les modèles et ces transformations de modèles forment le cœur de l'Ingénierie Dirigée par les Modèles (IDM). Les travaux réalisés dans le domaine de l'IDM ont permis la proposition d'outils qui permettent aux architectes et aux développeurs de modéliser les différentes perspectives du système qu'ils cherchent à construire et de les relier directement au code dans un langage de programmation d'une plate-forme particulière [P63]. L'utilisation de modèles en génie des processus offre les mêmes avantages. Les processus peuvent être développés en utilisant des approches de modélisation et affinés au fil du temps, selon les cycles de vie de développement de processus. Les techniques d'ingénierie dirigée par les modèles peuvent être utilisés pour le raffinement des modèles de processus.

6.2 Les limites des approches actuelles

Il existe de nombreux langages de modélisation de processus logiciels mais ils souffrent tous de trois limitations majeures :

1. une couverture limitée du cycle de vie du processus,

2. un manque de support pour la réactivité ou un mélange avec les flux de données qui rend les modèles difficiles à comprendre,
3. une modularité insuffisante qui rend difficile la réutilisation de modèles de processus.

Tout d'abord, ces langages semblent ignorer l'importance d'une approche cohérente pour modéliser un processus à toutes les étapes de son cycle de vie. En effet, soit un modèle unique représente le processus dans toutes les phases (par exemple on utilise le même modèle lors des phases de spécification et de réalisation), soit il faut transformer le modèle en une approche complètement différente pour l'exécution. Par exemple, le langage de modélisation de processus BPMN [P87] utilise une notation unique tout au long des phases de conception du processus et il n'offre pas la possibilité d'exécuter les modèles produits. Les développeurs de processus souhaitant allez plus loin et exécuter leurs modèles sont contraints de les transformer dans un autre langage (exécutable). Ainsi, il est courant de transformer des modèles BPMN en modèles BPEL [P83] pour les exécuter. La figure 6.1 représente la couverture du cycle de vie des processus de développement des différentes approches de l'état de l'art. Elle illustre le propos sur la couverture partielle des différentes approches existantes.

Deuxièmement, la plupart des approches se concentrent sur la définition de flux d'activités pour définir des ordres d'exécution possibles. Un désavantage général de l'utilisation des langages de modélisation de processus basés sur la notion de flux est le manque de dynamisme et de réactivité dans un tel modèle de processus. Cela rend ainsi très délicat la construction de modèle de processus agile alors que cette approche est devenue centrale dans bon nombre de domaines. Certaines approches comme [P4] ou BPMN utilisent des mécanismes basés sur des événements pour ajouter une réactivité à leurs modèles, mais leur attention reste sur le flux global des activités. SPEM [P86] non plus ne prend pas en compte la notion d'événement contraignant les modèles de processus à suivre une approche de flux. Mais, certaines de ses extensions, comme xSPEM [P18], ajoutent cette capacité et les modèles de processus sont enrichis avec des contrôles réactifs. D'autres, comme les *Event Process Chains* (EPC) utilisent les deux types d'entrées pour les activités (événements et artefacts) [P1]. Ainsi, si deux activités EPC doivent interagir, elles doivent utiliser des événements. Terminons en citant l'approche des *Workflow Management Systems* (WfMS) qui placent également la notion d'événements au cœur du modèle d'interaction et permettent de construire des modèles de processus dont les activités offrent une certaine réactivité [P61]. Si ces différentes approches permettent de représenter les deux formes d'interaction et donc supportent la définition d'activités réactives, cela encombre les modèles de processus qui deviennent complexes à comprendre. En effet, le flux de données et les flux de contrôle d'un processus étant représentés en même temps, il est difficile de conceptualiser les interactions exactes entre une activité et son contexte (les autres activités mais aussi les acteurs).

Enfin, l'absence d'une approche modulaire dans les méthodes de modélisation des processus limite la réutilisation des processus modélisés. Ainsi, même si la nature d'un processus est naturellement hiérarchique : il est composé de sous-processus ou activités, ce découpage n'est pas réellement adapté à la réutilisation. Or, les modèles de processus actuels utilisent des architectures de processus qui structurent les processus en suivant cette décomposition hiérarchique et sans s'intéresser aux interactions. Par exemple, SPEM propose la notion de composant de processus, mais les interfaces de ces composants sont limitées à représenter des produits. L'interaction avec un tel composant est donc limitée à un flux de données. Chaque composant de processus fournit donc ses produits uniquement lorsque ses entrées sont disponibles. Une encapsulation appropriée du processus devrait également limiter les interactions, y compris

les flux de contrôle et la chorégraphie à travers des interfaces spécifiées. MODAL améliore le concept de composant de processus offert par SPEM en utilisant la notion de ports qui offrent des services [P96]. EPC et WfMS n'offrent aucune solution spécifique pour l'encapsulation des processus. L'absence d'une approche modulaire et contractuelle dans les méthodologies de modélisation des processus limite la réutilisation des processus modélisés. Même si une réutilisation opportuniste des processus reste possible. La réutilisation systématique des modèles de processus n'est faisable que lorsque les processus sont conçus pour être réutilisés. De nombreuses applications de modélisation de processus actuelles, offrent la possibilité de stocker les processus dans des dépôts [P37]. Ces solutions offrent des fonctionnalités pour le stockage, la récupération et la gestion des versions. Le *process mining* peut également être utilisé pour sélectionner le meilleur candidat à la réutilisation [P2]. Mais trouver un procédé approprié pour la réutilisation n'est qu'une partie de l'effort. Il faut également intégrer cet élément à réutiliser dans son modèle de processus. Or, les approches de modélisation des processus n'offrent pas de moyens appropriés pour intégrer simplement un processus réutilisable dans un modèle de processus. Une approche modulaire conçue pour la réutilisation pourrait aider à résoudre ces problèmes d'intégration.

Dans ce cadre, nous avons proposé une méthode qui comble les lacunes des approches actuelles et permet aux concepteurs de processus d'exploiter le raffinement pour le développement de leurs processus. Pour cela, nous proposons une approche par phase et l'utilisation de la modélisation à plusieurs niveaux d'abstraction au sein d'une phase. Une telle approche permet de mieux maîtriser le développement progressif d'un processus et favorise également la réutilisation des processus d'une manière plus systématique. Nous avons proposé un cadre de modélisation de processus ainsi que des outils associés pour supporter notre méthode et ainsi résoudre les trois problématiques présentées plus haut.

6.3 Une approche multi-méta-modèle

Afin d'offrir, dans chaque phase du développement d'un processus, un langage adapté aux problématiques de la phase en cours, nous proposons d'utiliser un méta-modèle spécifique à chaque phase. Ainsi, pour chaque phase du développement d'un processus, il faut concevoir un méta-modèle distinct et spécialisé. Cela conduit à une famille de méta-modèles de processus qui traitent la modélisation du même processus dans ses différentes phases de développement. Le nombre de méta-modèles dépend du cycle de vie de développement de processus choisi et ce, sans limite fixée a priori, ce qui permet de s'adapter assez facilement aux pratiques en place. Dans le cadre de sa thèse Fahad Golra a illustré notre approche en utilisant quatre phases : la *spécification*, l'*implémentation*, l'*instanciation* et la *surveillance* de processus, comme le montre la figure 6.2. Chacune des phases de développement utilise un méta-modèle spécifique : un méta-modèle de spécification des processus, un méta-modèle d'implémentation des processus et un méta-modèle d'instanciation des processus. La phase de surveillance n'est pas une phase de développement, elle ne nécessite donc pas la définition d'un méta-modèle. Elle utilise un modèle d'exécution qui est généré à partir du modèle d'instanciation du processus puis mis à jour pendant sa réalisation par l'interpréteur de processus. Ceci permet d'exécuter et de surveiller les processus. Les utilisateurs qui souhaiteraient suivre notre approche peuvent choisir d'ajouter d'autre phases et de développer les méta-modèles nécessaires.

La figure 6.2 illustre également une autre originalité de notre approche : l'approche à deux niveaux de modélisation. Dans la couche de méta-modélisation, nos trois méta-modèles sont

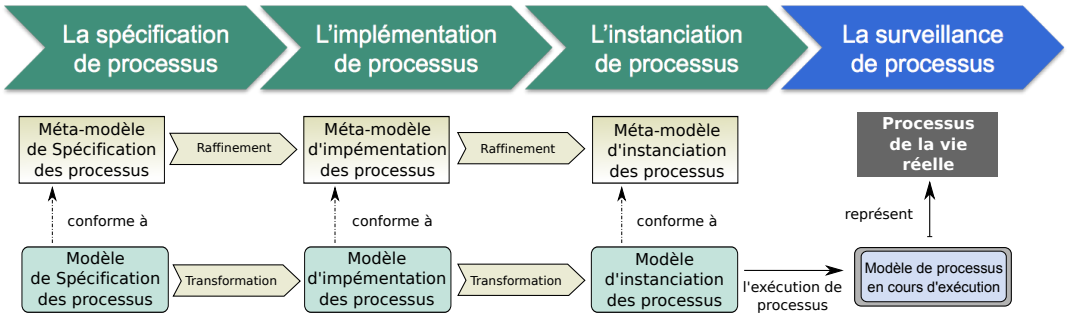


Figure 6.2 – Des méta-modèles multiples pour la modélisation de processus

représentés ainsi que les relations de raffinement qui les relient. En effet, nous défendons l'idée que chaque phase du développement raffine le (modèle du) processus en ajoutant des détails et choix spécifiques à la phase en question. Dans la couche de modélisation, les modèles de processus respectifs forment une chaîne dans laquelle on passe d'un modèle à l'autre par des transformations de modèles. Chaque transformation raffine le modèle de processus en injectant plus de détails et de choix de conception.

Le modèle de spécification du processus est développé en conformité au méta-modèle de spécification du processus. Celui-ci se concentre sur l'ensemble des éléments qui sont nécessaires pour décrire les exigences de développement du processus étudié (*Process Under Study*). Un processus de développement logiciel est spécifié à l'aide de ce modèle, et donc n'est pas surchargé par des détails d'implémentation. Le concepteur peut donc se concentrer sur les problématiques qui le concerne. Le niveau spécification peut être utilisé pour documenter des bonnes pratiques de l'entreprise ou du domaine en terme de processus en décrivant leur structure sous forme d'ensemble d'activités et de flux d'artéfacts. Ces descriptions restent cependant abstraites car elles ne décrivent pas comment les activités doivent être conduites, ni le projet qui est réalisé. Cela favorise la réutilisation de ces modèles de processus abstraits. De plus, les transformations définies rendent plus facile la réutilisation de ces modèles en permettant des les adapter au besoin du processus que l'on souhaite développer pour un projet et une entreprise spécifique. Ainsi, les standards de processus et les bonnes pratiques sont documentés de manière réutilisable. Le modèle d'implémentation de processus est conforme au méta-modèle d'implémentation. Ce modèle décrit les détails spécifiques à un projet et les intègre au modèle de spécification en ajoutant les détails de réalisation. Le méta-modèle d'implémentation est sémantiquement plus riche pour exprimer les choix de réalisation de chacune des activités ainsi que leur enchaînement. De cette façon, un modèle de spécification de processus unique peut être utilisé pour plusieurs implémentations dans le cadre de différents projets et ceux au sein de plusieurs entreprises. Enfin, le modèle d'instanciation de processus est développé et rend les processus exécutables en les reliant à des outils de développement, des documents, des dépôts, des personnes, etc.

6.4 Les méta-modèles pour le développement de processus

Les trois méta-modèles que nous avons définis pour démontrer l'applicabilité de notre approche dans le cadre d'un cycle de vie simple de développement de processus sont décrits rapidement dans cette section.

6.4.1 Le méta-modèle de spécification

Le méta-modèle de spécification permet de définir la structure de base du modèle de processus à l'étape de spécification. Un processus est ainsi défini comme une architecture d'activités, ces activités sont connectées et visent collectivement à atteindre un objectif commun. Une activité est une unité d'action dans un modèle de processus. Elle peut être primitive et ainsi non-décomposable (au niveau d'abstraction courant [1]) mais elle peut également être composite. Dans ce cas, elle contient un processus, ce qui revient à contenir une collection d'activités interconnectées. En plus d'offrir une hiérarchie de processus simple, notre approche permet de partager une activité entre plusieurs processus. Ainsi, il est possible de factoriser des actions de traitement communes à un ensemble de processus, qui partagent alors une activité.

Dans la conception par contrat (DbC), toutes les interactions de / vers un composant sont traitées par des interfaces spécifiées sous la forme de contrat qui expriment les exigences et les produits des composants. Notre approche de modélisation se place dans le cadre de la DbC, un contrat d'activité spécifie les artefacts d'entrée (requis) et de sortie (fourni) de cette activité. Ces descriptions d'artefacts sont des spécifications qui permettent de décrire des pré-conditions et des post-conditions de réalisation pour une activité. Les contrats de deux activités sont reliés par des liaisons. Une liaison relie le contrat fourni d'une activité au contrat requis d'une autre activité. L'artefact fourni par la première activité doit remplir les pré-conditions de la deuxième activité. Ceci définit un flux d'activités sur la base des artefacts qu'elles utilisent et produisent. Chaque activité définit des responsabilités pour son traitement. Chaque responsabilité est attribuée à un rôle ou à une équipe. Notre modèle de spécification de processus peut être traduit de ou vers d'autres approches de modélisation de processus existants. Cependant, cette traduction provoque en général une perte conceptuelle car elle est plus complète que la plupart des langages.

La figure 6.3 présente un exemple d'un tel modèle dans le cadre d'un processus standard de gestion du changement. Ce processus, dit ISPW, est couramment utilisé pour comparer les approches de modélisation de processus. Dans sa thèse, Fahad l'utilise pour présenter les différents méta-modèles que nous proposons. Ici, nous représentons uniquement deux activités *Conception* et *Revue* qui ont respectivement un contrat fourni et un contrat requis. La spécification du document de conception est présent à la fois dans les contrats fournis et requis. Les rôles et responsabilités de l'activité *Conception* sont détaillés : un rôle d'ingénieur de conception et deux responsabilités, signataire et responsable. Les interactions entre les activités sont représentées par l'utilisation des flots. Les post-conditions sont présentes dans le contrat fourni, par exemple *Conception* doit produire le document de conception avec la mise en évidence des modifications. Les pré-conditions sont présentes dans le contrat requis.

6.4.2 Le méta-modèle d'implémentation

Le méta-modèle d'implémentation met l'accent sur une séparation des préoccupations par la création de deux langages : un abstrait qui définit les activités et un concret qui contient les réalisations concrètes de ces activités. Au niveau abstrait, une (instance de) *ActivityDefinition* est un type d'activité alors qu'au niveau concret les (instances de) *ActivityImplementation* définissent un ensemble de réalisations alternatives de l'activité. La structure interne d'une activité est reléguée au niveau concret ainsi que ses propriétés qui sont encapsulées dans l'*ActivityImplementation*. Pour un processus, chaque *ActivityDefinition* provient du raffinement

1. Elle pourrait être remplacée plus tard par une activité composite.

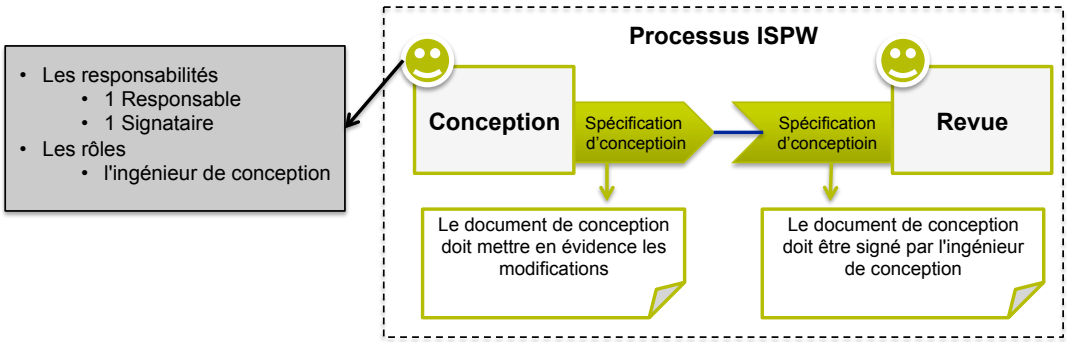


Figure 6.3 – Un exemple de modèle de spécification du processus

d'une activité du modèle de spécification du processus. De même, un contrat provient d'un contrat dans le modèle de spécification. Au niveau abstrait, les contrats sont spécialisés en trois contrats différents :

- Le contrat d'artefact décrit la spécification des artefacts en entrée et en sortie de l'activité en précisant leur méta-modèle.
- Le contrat de cycle de vie définit le cycle de vie de l'activité sous la forme d'un automate.
- Le contact de communication définit les messages entre les rôles associés aux activités.

Au niveau concret, les contrats utilisent des événements pour décrire les flots de contrôle. Ainsi, tous les événements (concrets) sont reliés soit à des artefacts, soit à l'automate du cycle de vie de l'activité ou soit à des messages. La chorégraphie réelle entre les activités et les rôles est donc définie par ces événements. Cela permet de séparer la définition du flot de données des activités de leur flot de contrôle. Ainsi, il est plus facile d'intégrer une gestion de configuration des artefacts puisque leur gestion n'est plus liée au flot de contrôle. Dans notre prototype, ses artefacts sont systématiquement manipulés à travers un dépôt de données versionnées [2]. La chorégraphie des activités est alors assurée par les événements de contrats concrets et peut donc être gérée efficacement par un système de gestion d'événements sous-jacent. Notons que la relation *implements* entre une implémentation d'une activité et sa définition est établie par une relation de conformité entre ses contrats concrets et les contrats abstraits auxquels ils correspondent.

Le modèle d'implémentation du processus de l'exemple précédent est représenté sur la figure 6.4. *Conception* est une définition d'activité du niveau abstrait et *Conception-Agile* est l'une de ses réalisations au niveau concret. Une définition d'activité peut avoir plusieurs implémentations comme ici *Conception-Agile* et *Conception-RAD*. Le flot entre les définitions d'activité *Conception* et *Revue* représente le flot de données du processus. Le flot de contrôle du processus est lui représenté par les flots possibles d'événements entre *Conception-Agile* et *Revue-Stratégique* par exemple. Les rôles sont associés aux implémentations d'activité au niveau concret et réalisent des responsabilités.

6.4.3 Le méta-modèle d'instanciation

Une fois le modèle d'implémentation construit, l'objectif du raffinement suivant est de rendre exécutable ce processus. Pour cela, on enrichit les détails des activités avec des informations sur

2. En l'occurence un dépôt subversion.

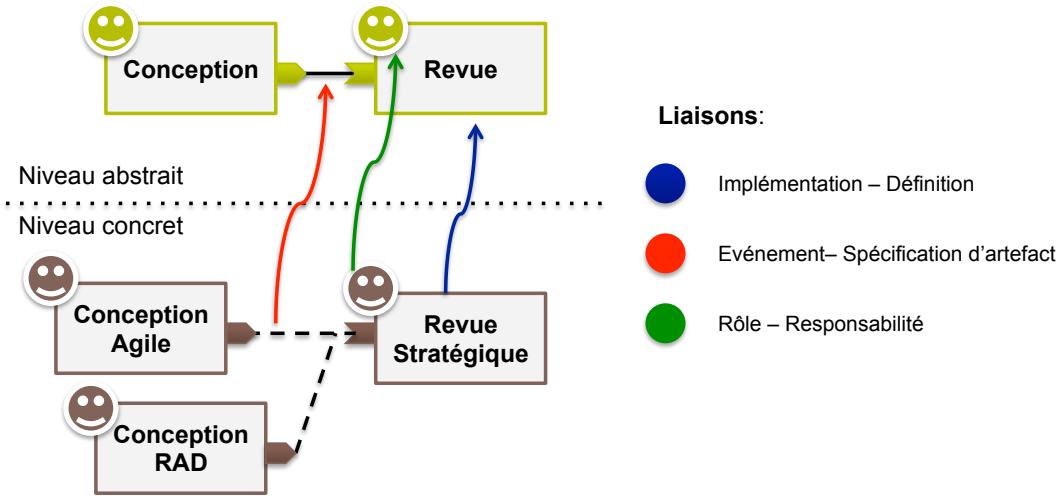


Figure 6.4 – L'exemple du modèle d'implémentation du processus

la durée, la date de commencement, l'état d'exécution actuel, les itérations et l'itération courante, etc. Comme avec les méta-modèles précédents, la structure du méta-modèle d'instanciation s'articule autour de la notion d'abstraction en séparant des définitions dans un niveau abstrait et des réalisations avec leur état dans un niveau concret. Ainsi, au niveau abstrait, les contrats définissent les spécifications des artefacts. Ces artefacts sont conservés dans un dépôt d'artefacts [3] et chaque artéfact dispose d'un identifiant unique (une URL dans notre prototype) qui est utilisé pour le manipuler. Le dépôt d'artefact supporte la gestion de versions ainsi que la pose (et le retrait) de verrous. Un artefact peut être verrouillé par une activité si sa nature le requiert, dans ce cas, les autres activités ne peuvent pas modifier l'artéfact. Une différence importante entre le méta-modèle d'implémentation et celui d'instanciation est le contenu d'une activité composite de niveau concret. Une activité composite du modèle d'implémentation ne contient que le processus concret avec les implémentations de l'activité qui sont liés aux définitions d'activité. Il ne contient pas les définitions de ces activités. À l'opposé, une activité composite du modèle d'instanciation contient à la fois le processus abstrait et le processus concret, ce qui signifie qu'elle contient un modèle complet du processus. Ainsi toute activité du modèle d'instanciation ne peut être liée à une définition d'activité hors de son contexte. La réalisation effective de chaque activité doit être choisie parmi les différentes réalisations disponibles. Cette réalisation devient active, les autres peuvent être conservées dans le modèle d'instanciation et seront alors passives. Lors de l'exécution du processus le choix de la réalisation active pour chaque activité peut être remis en cause et une autre réalisation (passive) peut devenir active. Cela permet d'adapter le processus pendant son exécution. Notons que dans notre prototype, nous ne supportons pas le chargement dynamique de nouvelles réalisations d'activité, le choix des réalisations (passives) incluses dans le modèle d'instanciation est donc crucial [4].

La figure 6.5 présente un modèle d'instanciation de notre exemple. Dans cet exemple, nous avons choisi d'utiliser *Conception-Agile* comme réalisation active pour l'exécution. Un flot de données lors de l'exécution entre deux activités signifie un transfert d'artefact entre elles via un

3. Un artefact peut être soit une copie papier ou un document numérique, ici nous nous limitons aux artéfacts numériques.

4. Une réutilisation des travaux du domaine du DSU pour modifier cette limitation serait sans doute intéressant.

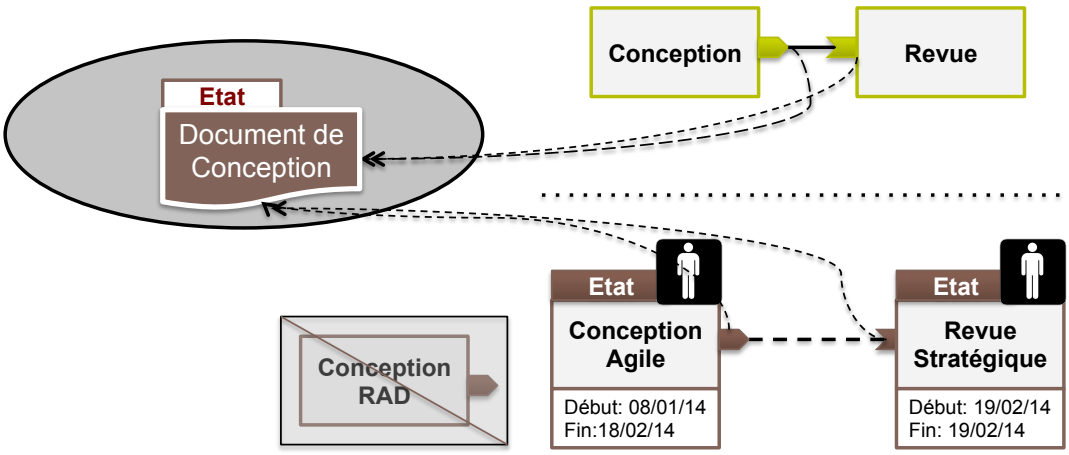


Figure 6.5 – L'exemple du modèle d'instanciation du processus

dépôt d'artefact. Par exemple, *Conception-Agile* produit le document de conception et le place dans le dépôt. *Revue-Stratégique* peut ensuite y accéder depuis le dépôt. Des événements de l'activité *Conception-Agile* seront diffusés contenant l'identifiant du document de conception lorsque c'est nécessaire (par exemple, lors de sa validation). Les états sont ajoutés aux activités et aux artefacts en fonction de leurs cycles de vie. Les propriétés pour la planification et l'organisation temporelle sont ajoutées au processus pour l'exécution. Par exemple, les dates de début et de fin pour chaque activité. Enfin, les acteurs sont ajoutés aux implémentations d'activité qui jouent les rôles spécifiques. Par exemple, Fabien peut jouer le rôle d'ingénieur de conception pour *Conception-Agile*.

6.5 Un environnement de spécification et d'exécution

Comme déjà mis en avant, un des problèmes avec les approches existantes de modélisation de processus, est qu'ils ont une couverture limitée du cycle de vie de développement d'un processus. Les concepteurs de processus ont besoin d'une approche pour la spécification de processus, puis de les transformer pour décrire leur implémentation et enfin de les transformer encore pour les rendre exécutables. Lorsque ces transformations requièrent un changement radical de modèle (par exemple le passage de BPMN à BPEL), elles provoquent des pertes sémantiques en raison des différences de sémantique des plates-formes de modélisation. Ces transformations provoquent aussi des ruptures qui rendent délicates la traçabilité et des approches agiles de définition des processus.

Nous avons construit un prototype qui permet d'appliquer l'approche que nous avons proposée. Son architecture est présentée en figure 6.6. La partie gauche de ce schéma décrit des outils pour développer les modèles de processus des différents niveaux. Les modèles de processus pour le niveau de spécification sont développés en utilisant un éditeur de processus graphique (fait en utilisant Openflexo) ou à travers un langage de domaine spécifique textuel (réalisé en XText). Ces modèles de processus peuvent être utilisés pour développer des standards de processus que ce soit au niveau de l'entreprise ou à un niveau plus large. Plusieurs standards peuvent être développés dans cette phase, ils sont ensuite stockés dans le dépôt pour une utilisation future. Les modèles de spécification d'un ou plusieurs processus sont ensuite transformés en des modèles de processus

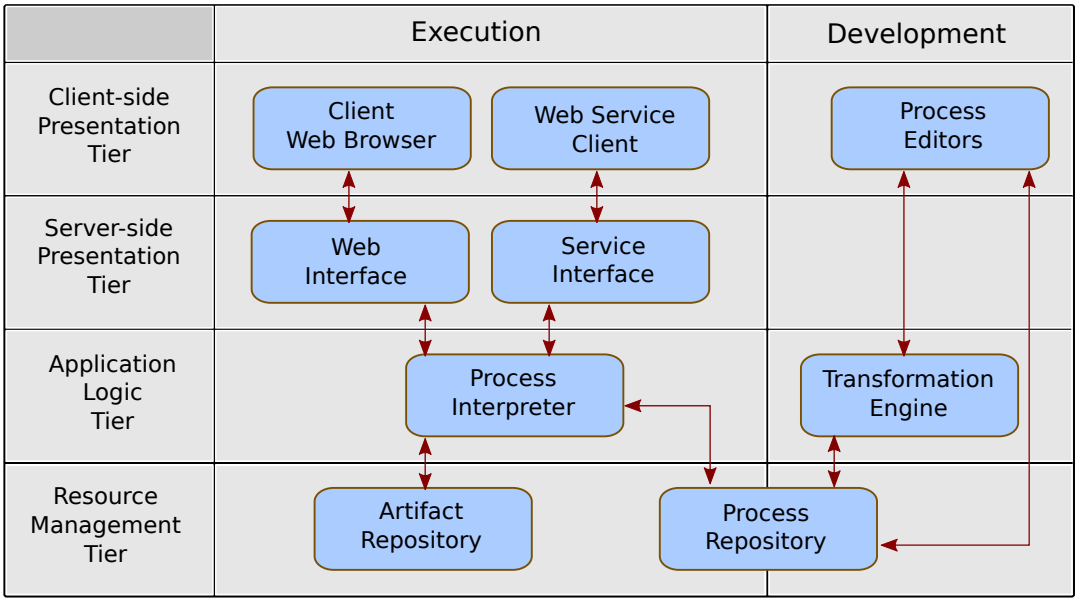


Figure 6.6 – L'architecture de notre prototype

d'implémentation en utilisant un moteur de transformation, fourni avec le prototype. Ce moteur de transformation transforme les spécifications de processus en des implémentations initiales. Le développeur du processus peut alors raffiner manuellement ce modèle d'implémentation pour y injecter des informations et des choix. Plusieurs implémentations d'une même activité peuvent être définies, ainsi, l'entreprise (ou l'organisation) peut se constituer une base d'activités usuelles avec les différentes implémentations. Plusieurs modèles de processus de spécification peuvent être utilisés pour développer un seul modèle d'implémentation. Ceci est utile lorsqu'un processus doit être conforme à plusieurs pratiques / normes / standards. Une fois le modèle terminé, une nouvelle transformation peut être appliquée pour produire un modèle d'instanciation initial qu'à nouveau le développeur va pouvoir configurer. Une fois qu'un modèle de processus devient exécutable, il peut être chargé dans l'interpréteur de processus du prototype (partie gauche de la figure). L'interpréteur de processus créé les instances du modèle de processus et les exécute. Une interface web, appelée tableau de bord de gestion de projet, permet d'interagir avec les processus pendant leur exécution. Quelques captures d'écran sont présentées en figure 6.7. Le propriétaire d'un processus peut accéder à toutes les activités de ses processus. Un acteur joue un rôle spécifique pour chaque activité, ce qui détermine ses droits d'accès ainsi que les fonctionnalités qu'il peut utiliser pour chaque activité. Une fois que les artefacts requis par une activité sont disponibles, les acteurs associés à cette activité peuvent télécharger les artefacts dont ils ont besoin. Ils peuvent alors réaliser les actions correspondant à l'activité puis téléverser les artefacts développés ou modifiés à travers cette interface web. Le propriétaire d'un processus peut adapter une de ses activités en changeant son implémentation active en une des implémentations alternatives présentes. Il doit alors s'occuper du transfert de l'état entre l'ancienne et la nouvelle implémentation de l'activité. Ce transfert d'état peut être géré par l'interpréteur de deux manières :

1. en utilisant des fonctions de transfert prédéfinies dans le comportement de la nouvelle réalisation d'activité,

2. en présentant les propriétés des deux implémentations au propriétaire du processus qui peut réaliser le transfert de l'état à travers un formulaire.

Notons que dans notre approche, chaque étape de raffinement maintient des liens de traçabilité avec les précédentes. Ainsi, il est plus facile de remettre en cause des choix réalisés et il devient possible de rattacher les activités et artéfacts d'exécution aux pratiques / normes / standards choisis pour le processus.

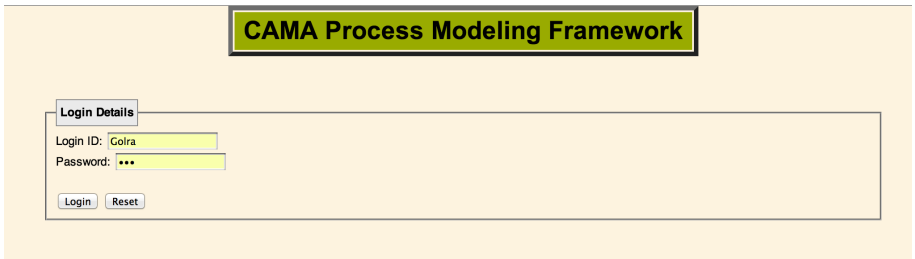
6.6 L'évaluation de l'approche

Pour valider notre proposition, nous avons utilisé un cas d'étude et une évaluation basée sur la réalisation d'un ensemble de patrons d'activité (*workflow patterns*).

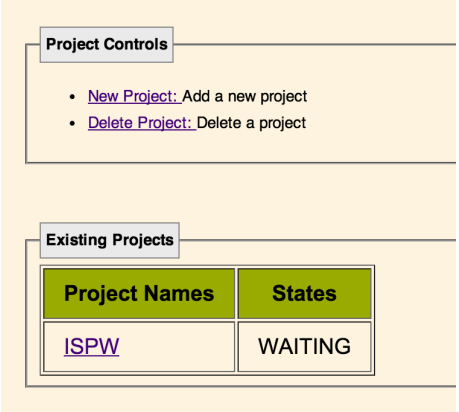
Le cas d'étude porte sur une entreprise fictive, *TB-Enterprise* et présente le développement complet d'un processus pseudo-réel. Le scénario choisi pour cette étude de cas est pseudo-réel car le processus original est légèrement modifié pour illustrer les concepts clés de notre approche. *TB-entreprise* développe un processus pour organiser le test d'un système appelé *AlphaSystem*. L'entreprise suit notre approche, elle développe donc un ensemble de modèles du processus, chaque modèle correspondant à une phase spécifique du cycle de vie du processus. Ce cas d'étude montre comment ces modèles sont raffinés. Il est également utilisé pour étudier la conformité à plusieurs standards de processus. Ce cas d'étude démontre également la façon dont les acteurs interagissent avec l'exécution des modèles de processus à travers le tableau de bord de gestion de projet. Comment leurs droits d'accès à certaines activités et certaines actions liées à une activité spécifique sont gérés par le tableau de bord. Il est décrit en détail dans le chapitre 6 de la thèse de Fahad [P49].

Il existe trois principales préoccupations dans un modèle de processus : son flot de contrôle, son flot de données et ses ressources. Un ensemble de travaux de recherche et d'analyse ont conduit à l'élaboration d'un catalogue de patrons sous la bannière *Workflow Patterns Initiative*[5]. Le but de cette initiative est de fournir un ensemble de pratiques courantes pour permettre une évaluation plus facile des différentes propositions de langage de modélisation de processus. Les résultats de l'évaluation de nombreuses approches et outils de modélisation de processus sont fournis par l'initiative et permettent donc une étude comparative. Il est proposé 40 patrons pour les flots de données, 43 sur les flots de contrôle et 43 sur les ressources. Nous avons, dans la mesure du possible, réalisé ces patrons dans nos langages de modélisation pour les comparer avec les approches classiques en terme d'expressivité. Les résultats de cette étude exhaustive sont résumés par le graphique de la figure 6.8. Nos résultats sont comparés avec les approches traditionnelles : les diagrammes d'activité d'UML, les approches de gestion de processus métier BPMN et BPEL et une approche de gestion de *workflow* COSA. Les résultats de la comparaison avec les autres approches sont très encourageants dans les trois axes. Nous sommes en mesure de supporter complètement et partiellement plus de 80 % des patrons dans les trois catégories. De plus, les patrons qui ne sont pas supportés par notre approche ne le sont pas en raison de défauts de la méthode. En effet, les raisons de l'absence de support sont soit un choix de notre implémentation (qui pourrait donc être étendue) ou les limitations du prototype (qu'il faudrait étendre). Par exemple, l'un des patrons requiert le transfert d'artefacts par valeur entre les activités. Or, nous avons choisi de réaliser tous les transferts par référence en utilisant un dépôt d'artefacts dans notre prototype d'interpréteur. Ce qui ne peut clairement pas être considéré comme un défaut de la méthode. Le chapitre 7 de la thèse de Fahad [P49] contient une analyse

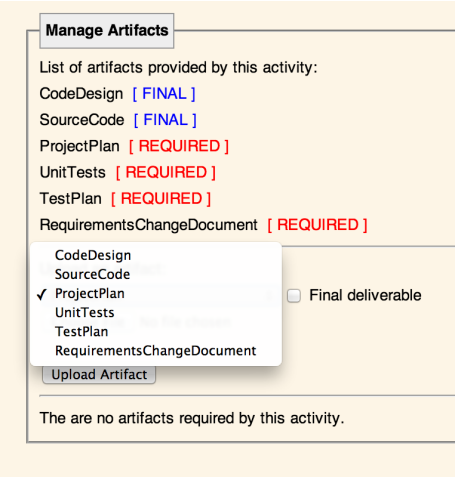
5. http://www.workflowpatterns.com



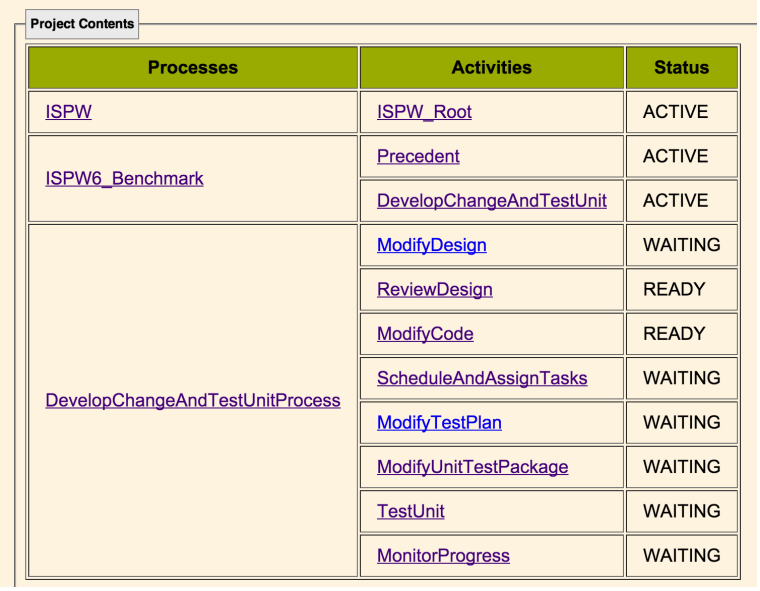
(a) Authentification de l'acteur



(b) Tableau de bord



(c) Détail d'une activité



(d) Les activités et leur état

Figure 6.7 – Des captures d'écran du tableau bord

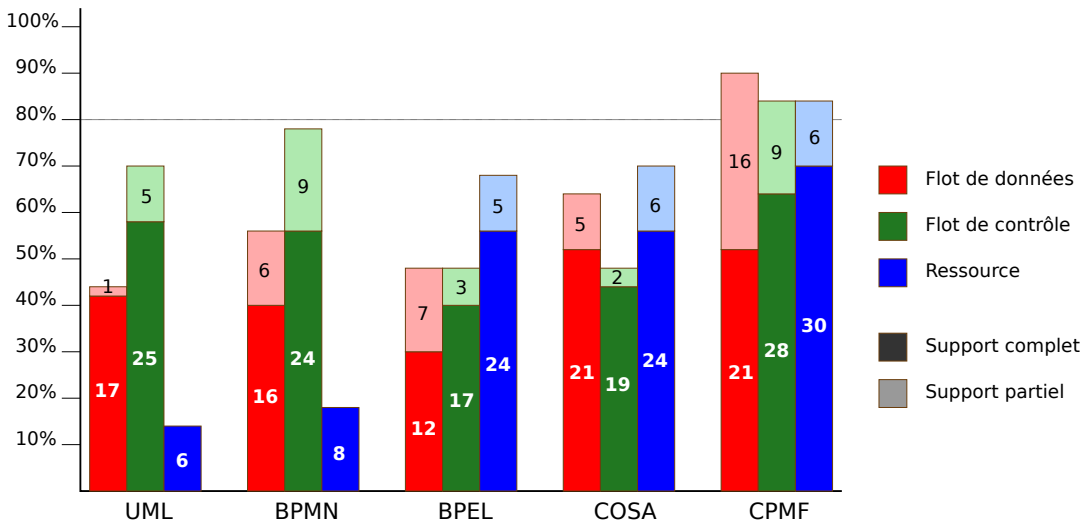


Figure 6.8 – Évaluation basée sur les patrons de *workflow*

plus détaillée et l'annexe A contient la description exhaustive des patrons et de leur réalisation dans nos langages.

6.7 Bilan et perspectives

Les travaux que nous avons réalisés avec Fahad peuvent être résumés en suivant l'idée d'Osterweil suivant laquelle *les processus sont des logiciels*. Il s'est agi pour nous d'adopter et d'adapter des approches qui ont fait leur preuve dans le domaine du génie logiciel. Ainsi, mettre au cœur de l'approche les notions de modèle, raffinement et réutilisation nous semble crucial. De plus, nous pensons que les approches processus actuelles font souvent peur aux entreprises [6] car :

1. Elles sont souvent très documentaires : le seul résultat de la définition (en avance) des processus est en général sa documentation. Même si cela plait aux ingénieurs qualité et facilite un peu l'intégration de nouveaux ingénieurs au sein d'une entreprise, cela reste assez rébarbatif pour les ingénieurs qui réalisent la plupart des activités.
2. Les approches actuelles sont peu agiles. Une fois un processus défini, il est difficile de remettre en cause des choix. Cela nécessiterait de refaire le modèle et la documentation associée. Entre le coût induit et les bénéfices relativement maigres que l'on peut en espérer, les processus réels divergent souvent par rapport à leur modèle original.
3. Aucun outil ne permet de surveiller la conformité réelle d'un processus en cours de réalisation par rapport au modèle qui en avait été fait.

Cette première incursion dans le domaine de la modélisation des processus avait pour but d'essayer de diminuer ces critiques en :

– permettant l'utilisation des modèles de processus pour surveiller la réalisation projet mais aussi pour aider à sa réalisation en diffusant l'information de manière plus fluide ;

6. Au moins à celle dans lesquelles, je suis passée ou avec qui j'ai collaboré.

– autorisant l'adaptation dynamique des activités d'un processus, ce qui permet de s'adapter aux évolutions du quotidien comme par exemple, les abandons de tâches ou les absences de personnel.

Nous pensons que la proposition faite permet d'aller dans le bon sens même si elle reste préliminaire et qu'il faudrait la valider en pratique au sein d'entreprises. Nous serions ainsi à même de déterminer les qualités et défauts réels de notre approche.

Pour poursuivre sur ce chemin, mes travaux actuels avec Openflexo sur la fédération de modèles et ceux sur l'ingénierie des besoins dans le cadre du projet ANR Formose, qui débute, visent à proposer de mieux relier l'interpréteur de processus aux outils d'ingénierie. Il s'agirait ainsi de fournir des guides aux ingénieurs au sein même des outils qu'ils utilisent au jour le jour. Ici, on ne veut pas seulement mieux intégrer ses outils avec la gestion de configuration ou la gestion de la traçabilité mais plus généralement proposer des mécanismes de chorégraphie des différents outils. Cette chorégraphie serait utile aussi bien au niveau individuel dans sa pratique quotidienne que pour aider à la collaboration avec les autres acteurs. Notons que cette approche devient une problématique générale dans le domaine, les différents outils cherchant à s'interfacer à différent niveau. Ainsi, on peut citer les efforts d'IBM autour de la plate-forme Jazz et des outils *team concert* ou OSLC qui vise à devenir un standard d'interactions entre les différents outils qui sont utilisés pour gérer le cycle de vie du développement d'un produit ou d'une application. Un autre domaine dans lequel, on retrouve de manière pragmatique cette problématique est celle de l'intégration continue utilisée de plus en plus lors de la production de logiciels. On peut, enfin, également citer des outils comme Github qui cherche à devenir un outil de gestion du développement qui peut s'accommoder de toutes les pratiques et peut se connecter à de nombreux outils grâce une politique d'API assez ouverte. Ainsi, que ce soit par des scripts automatisés utilisant des API ou par des modèles d'intégration, on constate un accroissement du couplage entre le processus de développement et les produits du développement (comme le code par exemple).

Chapitre 7

Bilan et perspectives

Chacun des chapitres de ce manuscrit s'ouvre sur un résumé des contributions qu'il décrit et se termine sur une conclusion qui ouvre les perspectives de ces différentes propositions. Le but de cette conclusion va donc être, plutôt que de les reprendre et les énumérer, de synthétiser quelques idées que ses travaux ont initiées ou renforcées. Les perspectives seront alors l'occasion de montrer comment j'envisage de continuer mes travaux de recherche ainsi que leurs articulations avec ces idées.

7.1 Bilan

Tout d'abord, il convient de résumer quelques idées scientifiques sur la notion de modèle qui est clairement le centre de mes activités.

1. **Tout est modèle.** Lorsque l'on *manipule* du logiciel que ce soit pour le concevoir, pour le réaliser, pour le valider, pour le déployer, pour l'utiliser, pour le faire évoluer, il est nécessaire de traiter des artéfacts de formes diverses. Chaque communauté de pratique repose sur ses formes d'artéfacts : modèles graphiques pour la conception, fichiers de code pour la réalisation, fichiers de configuration pour le déploiement, *commit* dans un dépôt `git` pour l'évolution par exemple. Il me semble que, bien qu'importante, la forme de représentation et de sérialisation de ces artéfacts est moins importante que les informations qu'ils contiennent. En tant que conteneurs d'informations sur le logiciel, ils sont des modèles du logiciel. De plus en plus, j'essaye donc de définir des méta-modèles pour représenter ces éléments tout en essayant de les maintenir connectés aux artéfacts auxquels ils correspondent. Je travaille donc à produire des outils pour aider à la gestion de modèles / méta-modèles pour cette raison. Je suis leur premier client.
2. **La représentation d'un modèle et les modes d'interactions avec un modèle doivent être adaptés à un domaine.** Les concepts et informations contenus dans un modèle peuvent être génériques et utilisables dans des contextes divers. En revanche, la forme de représentation des ces informations doit être utilisable *facilement* dans un domaine. Elle doit donc être adaptée à un domaine. Ici domaine est pris au sens large, cela peut être un domaine d'application aussi bien qu'un métier différents. Ainsi, la conception d'un logiciel pour le web ou celui pour contrôler un véhicule vont utiliser des formes adaptées. Il en va de même pour les modèles destinés à la réalisation, à la gestion de configuration ou au déploiement du même logiciel. Au delà, de la seule représentation qui doit être adaptée aux usages de ses producteurs et de ses utilisateurs, il faut également

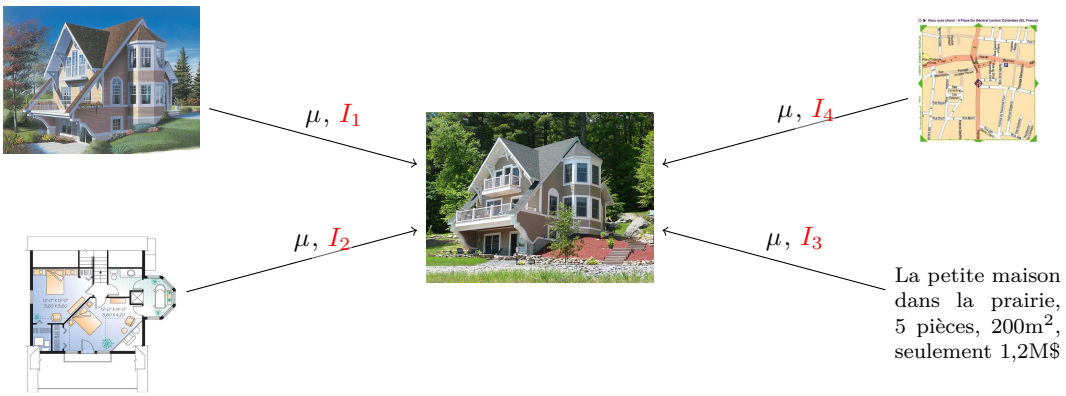


Figure 7.1 – De nombreux modèles aux formes diverses

que les opérations possibles sur ces informations correspondent assez naturellement à leurs actions. En effet, si le modèle ne leur convient pas des stratégies de contournement vont se mettre en place. Ces stratégies individuelles rendent délicates toute collaboration et pérennisation des pratiques. Ainsi, un modèle permet de calculer, d'imaginer, d'évaluer, d'analyser, de comprendre, de communiquer...

3. **Un modèle doit se focaliser sur un petit nombre de préoccupations.** Chacun de ces modèles doit être compréhensible pour pouvoir être utilisé facilement. Dans ce cadre, il est bon de ne pas y ajouter de l'information inutile. Dans ce cadre, je pense qu'il est plus adapté de construire plusieurs modèles comportant éventuellement des redondances entre eux plutôt que de vouloir adresser plusieurs préoccupations avec un unique modèle. Ainsi, mes travaux sur les modèles de composant, de déploiement ou de processus m'ont confortés dans l'idée de fournir plusieurs *petits* méta-modèles. Une des conséquences est qu'il faudra assurer une certaine cohérence entre les divers modèles d'une entité. Mais, en échange, les opérations sur le modèle peuvent trouver toutes les informations dont elles ont besoin facilement.
4. **Un modèle n'est jamais isolé, il prend son sens en relation à d'autres modèles.** Une des conséquences du point précédent est que pour manipuler un logiciel complexe, il va falloir de nombreux modèles car de nombreuses préoccupations entrent en jeu. Ces différents modèles ne sont pas indépendants, leurs contenus doivent être cohérents. Comprendre le logiciel sous-jacent exige d'utiliser un grand nombre de ces modèles pour couvrir suffisamment de préoccupations.
5. **L'intention et l'usage du modèle doivent être clair.** Un modèle étant un vecteur de communication entre son producteur et son utilisateur, il faut que l'intention du modèle et les utilisations que l'on peut en faire soient les plus claires possibles. Dans ce cadre, la forme de la représentation ainsi que l'ergonomie des outils de manipulation sont cruciales. Cela impose aussi de les documenter...

Pour illustrer, ces différents points j'utilise souvent les deux schémas des figures 7.1 et 7.2. Le premier illustre le fait que les modèles d'un système sont nombreux. Ici, on réutilise les notations classiques du domaine μ pour indiquer que quelque chose *modélise* autre chose. Associée à cette relation, il doit y avoir une intention (notée I et en rouge dans la figure). Les exemples de modèles en partie gauche sont choisis pour illustrer la diversité des formes de représentation. La

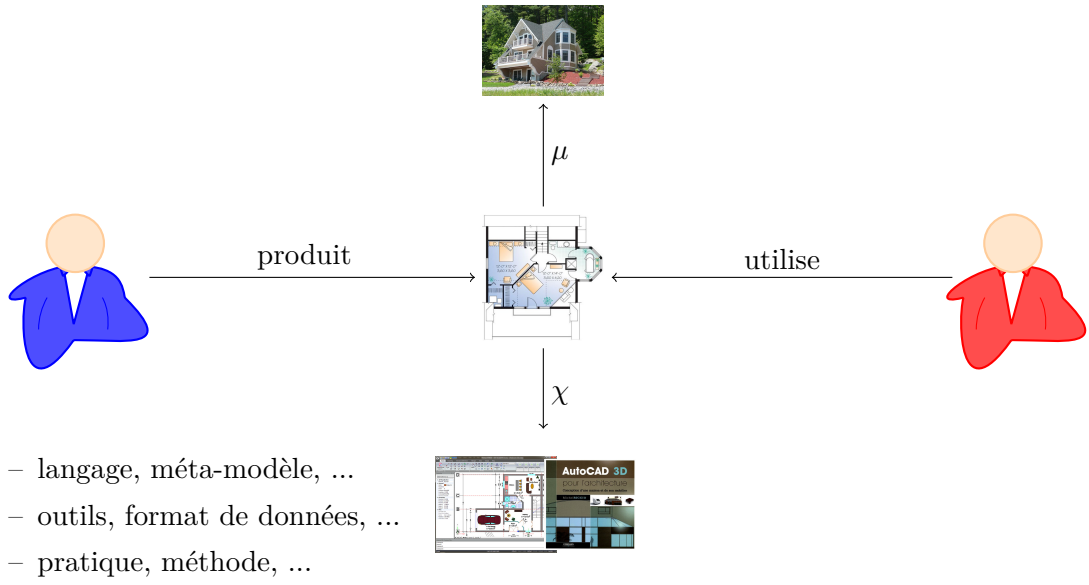


Figure 7.2 – Les modèles comme outils de collaboration

seconde figure illustre le fait que les modèles sont des outils de collaboration. Un modèle permet à un producteur d'exprimer de l'information sur le système que le destinataire du modèle peut utiliser. Pour cela, le modèle est exprimé dans un langage (on utilise ici la relation χ de conformité à un méta-modèle), en utilisant des outils et des pratiques. Ces éléments doivent bien sûr être partagés pour permettre la collaboration.

Ensuite, je tire également quelques enseignements sur le monde de la recherche et le métier de chercheur.

1. Il est important de couvrir tout le spectre allant de la description mathématique du modèle, de sa preuve (mécanisée de préférence), de son implantation à son utilisation dans des expériences. En effet, un modèle isolé ayant peu de sens, il peut difficilement être réutilisé par d'autres. De plus, en explorant à la fois des aspects formels et la mise en pratique, la *complétude*[1] du modèle est mieux assurée. Le travail formel et mathématique permet de se concentrer sur la minimalité et la précision. La réalisation et l'usage garantissent l'utilisabilité et la couverture. Ainsi, dans mon travail sur les dépendances et aussi plus récemment sur la mise à jour à chaud, j'ai essayé d'aborder tous ces aspects.
2. La recherche ne peut pas s'isoler du monde et de sa complexité. L'utilisateur ou l'industriel, même exigeant, ne peut pas être éliminé. Il est important de puiser les motivations des travaux de recherche dans le monde qui nous entoure. Il me semble tout aussi important de s'en servir pour valider les résultats. Ce travail parfois ingrat et souvent difficile permet souvent de mieux comprendre l'intérêt réel des propositions que l'on fait et de faire l'effort de les expliquer. Même si dans mon travail, je n'ai pas toujours réussi à aller jusqu'au bout par manque de temps, des bons interlocuteurs ou d'ingénieurs, cela reste un de mes objectifs majeurs.
3. La recherche doit s'inscrire dans la durée et dans la collaboration. Explorer réellement un problème, tester différentes solutions, valider les résultats sont des tâches qui prennent du

1. La complétude peut toujours être discutée, ici il s'agit d'exprimer une plus grande complétude possible.

temps. Les premières propositions s'avèrent souvent inadaptées lorsqu'on les présente à d'autres. Confronter permet donc d'avancer. Et c'est vraiment au cœur des discussions, parfois enflammées, que les argumentations se construisent [2]. Il est également important de revenir sur ce que l'on avait déjà proposé pour le voir sous un angle différent et éventuellement mieux en appréhender les limites. La rédaction de ce manuscrit a participé à ce mouvement en me forçant à faire une pause et à regarder en arrière.

4. Le monde de la recherche actuelle est difficile. Le financement est devenu rare, la compétition est devenue le maître mot et le contrôle doit s'exercer. Tous les chercheurs sont entrés dans une course sans fin au projet, à la publication, à la bonne évaluation... Cela entraine une multiplication des tâches à réaliser qui transforme cette course en une lutte contre le temps qui file. Cette tendance à l'accélération et la compétition s'oppose totalement au point précédent sur le rythme nécessaire à la recherche. Le chercheur se retrouve donc au cœur d'un dilemme et fait du mieux qu'il peut (c'est-à-dire parfois mal).
5. En tant qu'encadrant de projets, stagiaires, doctorants et post-doctorants, il faut être flexible pour s'adapter aux différentes personnalités et approches. Certains veulent être dirigés quand d'autres veulent être conseillés et qu'encore d'autres adorent être défiés. Sur les cinq doctorants que j'ai encadré, il y a cinq nationalités et cultures. Les pratiques étaient donc vraiment différentes. J'espère quand même leur avoir apporté et permis leur entrée dans le monde de la recherche même pour ceux qui n'ont pas été jusqu'au bout de leur thèse.

En terme de publications, la première partie de la bibliographie page 133 de ce document contient toutes mes publications classées.

7.2 Perspectives

J'ai de nombreuses idées de projet de recherche et j'en aurais probablement d'autres par la suite. Je vais néanmoins me limiter ici à la présentation de trois axes qui me semblent à la fois importants et naturellement émerger de mes travaux.

7.2.1 Fédération de modèles

La première perspective a déjà été évoquée dans l'introduction de ce document. Il s'agit de la fédération de modèles. L'idée est de formaliser les conséquences des différents points présentés dans la section 7.1. Le projet a déjà commencé avec Antoine Beugnard et mes collègues d'Openflexo Christophe Guychard et Sylvain Guérin. Nous essayons de construire des *langages* pour la modélisation. Ici les maîtres mots sont flexibilité et consistance. Pour cela, nous proposons de :

- supporter la *modélisation libre* ; dans ce cadre, un utilisateur doit pouvoir ajouter au modèle des éléments qu'il n'est pas encore capable de décrire complètement ni de classifier (de leur donner un type) ; un élément du modèle doit pouvoir *être instance de* plusieurs concepts et pouvoir changer de type ; il faut également pouvoir définir à tout moment de nouveaux concepts (par exemple, à partir d'une (ou plusieurs) instance(s), l'utilisateur peut identifier

2. A cette occasion, je ne peux que remercier Antoine, Christophe et Sylvain pour tous ces cafés ou séances devant un tableau qui ont duré des heures... Elles me sont chères car elles me semblent être l'essence même de ce qui nous nourrit en temps que chercheur.

un nouveau concept) ; enfin, il convient de pouvoir modifier la représentation de chaque instance ou concept.

– permettre la création de *modèles fédérés* ; les modèles ne sont plus construits de manière isolée mais reliés entre eux et maintenus en cohérence ; cela passe bien entendu par un support outillé pour aider les concepteurs de modèles dans le maintien de leur cohérence ; dans ce cadre, nous proposons de mieux séparer le stockage des informations (unitaires) de leur utilisation dans les modèles ; ainsi, un modèle n'est plus qu'un assemblage structuré d'informations, ces informations pouvant être contenues (sérialisées) soit dans le modèle, soit dans d'autres modèles, soit même à l'extérieur dans un conteneur adapté (une base de donnée, un fichier, ...) ; on associe aux liens entre les modèles des comportements qui permettent de gérer la cohérence lors de l'évolution de l'information.

Une illustration de ces points est l'outil *FreeModelingEditor* d'Openflexo présenté en figure 7.3 page suivante. Dans cet outil, un utilisateur peut créer des concepts et des instances. La partie gauche de l'IHM permet de lister les concepts, de connaitre leurs instances et de créer de nouveaux concepts. La partie centrale permet de créer des instances en leur associant une représentation graphique. Les instances peuvent être *instance de* un concept ou pas. Cette relation est multiple et dynamique. À tout moment, l'utilisateur peut la modifier. La partie droite permet de travailler sur la représentation graphique des instances. Un concept peut avoir une représentation graphique ou non et ses instances peuvent suivre cette représentation ou avoir leur propre représentation. Enfin, les propriétés des instances peuvent être reliées à des sources d'information extérieures au modèle comme par exemple d'autres modèles ou des fichiers externes [3].

L'utilisation principale de ce type d'outil est de construire des modèles dans des domaines sans pratique de modélisation établie. Ainsi, il est possible de co-construire des modèles et leurs méta-modèles.

Ces travaux ont déjà donné lieu à quelques publications [7, 22, 27, 8]. Ils ont été en partie réalisés dans les différents composants logiciels de l'infrastructure de modélisation que propose Openflexo. Enfin, leur application dans un cadre industriel plus important est prévu dans le cadre du projet ANR Formose en cours.

7.2.2 Des outils transverses pour le déploiement

Dans un monde plein de modèles, la construction d'un logiciel devient un enchainement complexe de tâches sur des modèles variés. Je suis convaincu qu'il est nécessaire d'offrir une aide à la fois en terme de théories, de méthodes et d'outils au développeurs d'application. En effet, les contraintes importantes pour le déploiement d'un logiciel peuvent apparaitre à toutes les étapes d'un développement logiciel. Il faut donc collecter ces contraintes sans qu'elles ne contraignent trop la réalisation ou le déploiement réel chez le client. Le développeur doit être assisté dans cette tâche par des outils qui l'aident à s'assurer de la complétude des contraintes qu'il spécifie. L'exemple de la gestion des importations de bibliothèques de Nix est, dans ce cadre, un exemple de solution. Il faut également aider à la transmission de ces contraintes et à leur raffinement lors du processus de développement. Dans ce cadre, la fédération de modèles déjà évoquée me semble pouvoir apporter une contribution. Mais il faudra sans doute aussi

3. Dans la version actuelle, il est déjà possible de se connecter à des fichiers tableurs (xls), textes (docx), des ontologies (owl), EMF et XML. Tout développeur peut réaliser un connecteur pour d'autres sources de données.

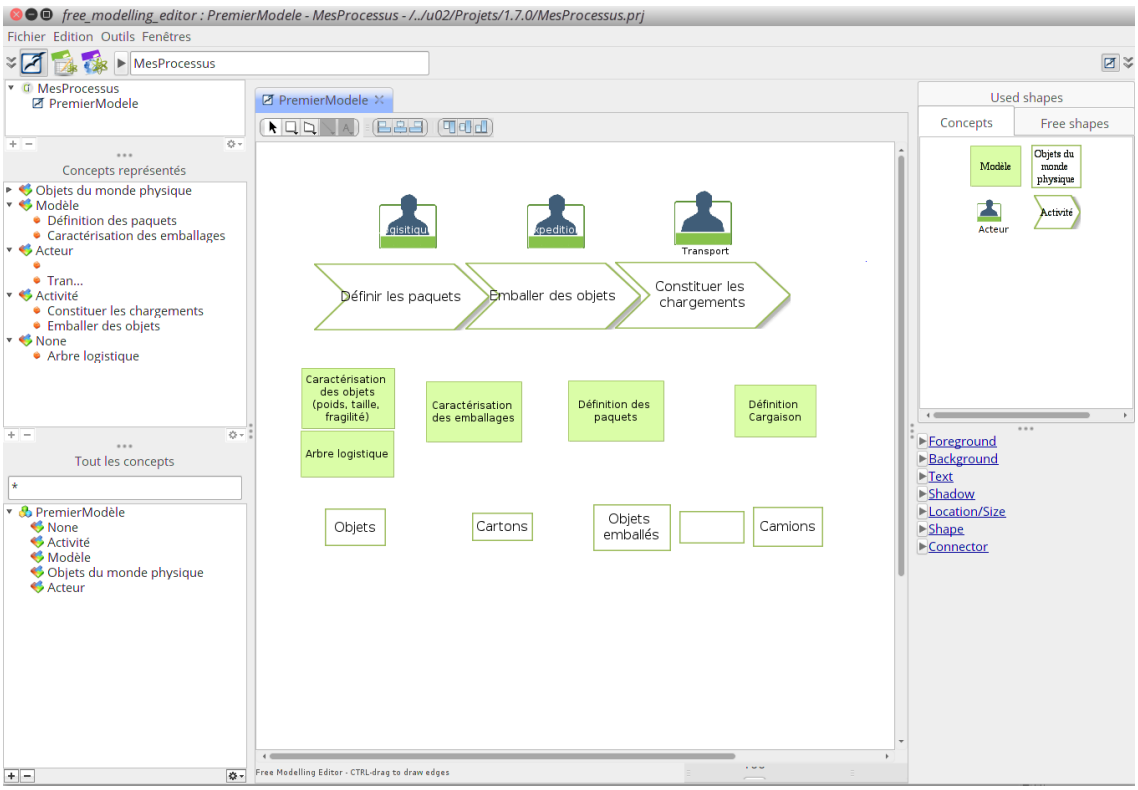


Figure 7.3 – Capture d'écran du *FreeModelingEditor*

définir des solveurs de contraintes et des outils de simulation de déploiement. Ainsi, ici, il s'agit de consolider et d'élargir les différents travaux déjà réalisés en utilisant une approche fédérative.

Parmi les points qui me semblent importants dans cet objectif, je voudrais citer les besoins en traçabilité des processus suivis. On rejoint ainsi les travaux présentés au chapitre 6. Seules ces traces permettraient de garantir des propriétés sur les exigences de déploiement construites pendant le développement. Un exemple de telle trace nécessaire est la façon dont le binaire a été compilé et en utilisant quelles versions de librairie [4]. Bien sûr tout cela repose sur une gestion de configuration de tous les *modèles* qui contribuent à la réalisation.

Pour la réalisation du déploiement, de nombreux efforts restent également à faire de façon à permettre la collaboration des nombreux outils de déploiement réel avec leurs périmètres bien définis. En effet, chacun ne peut résoudre qu'un sous-ensemble de toutes les contraintes de déploiement et il semble illusoire d'imaginer collecter toute l'information pour raisonner. Dans ce cadre, les travaux présentés dans ce document pour la gestion des dépendances paraissent naïfs. D'un point de vue théorique, je suis intéressé par la façon dont on peut répartir le raisonnement et déduire des propriétés globales. En particulier, dans un cadre où les solveurs locaux suivraient des logiques différentes. Il faut alors probablement qu'en plus de fournir une réponse, ils soient capables de la qualifier en décrivant ce qu'ils garantissent. Une approche à la Hets [5] peut-elle fournir des éléments de réponse ?

4. En ce moment, avec la divergence `gcc` versus `clang`, ce problème se pose de plus en plus sur les Mac...
5. http://theo.cs.uni-magdeburg.de/Research/Hets.html

7.2.3 Vers un déploiement dynamique transparent

Enfin, je voulais terminer ce document par la mise à jour à chaud (DSU pour les anglophones). Je pense que nous sommes toujours dans le début de l'utilisation de la mise à jour à chaud. Il existe de nombreuses plates-formes incompatibles ayant des propriétés variées mais peu de travaux sur les mécanismes sous-jacent qu'elles proposent. Dans ce cadre, les travaux initiés avec Sébastien Martinez et Jérémy Buisson sur la classification, la description et l'analyse systématique de ces différents mécanismes et leurs propriétés, permettront sans nul doute de progresser. Des modèles théoriques existant (comme pour les continuations) ou nouveaux pourront alors être étudiés. On peut alors envisager de construire des langages et des systèmes d'exécution dans lesquels la mise à jour à chaud sera une fonctionnalité de base et non pas un ajout a posteriori.

Plus généralement, il semble important de pouvoir contrôler un logiciel en exécution (pour le surveiller, l'aider, le suspendre, le modifier, ...). Dans ce cadre, les évolutions du domaine des langages vers l'abstraction amène à un écart de plus en plus grand entre chaque *instruction* qu'un programmeur utilise et les actions que va réaliser le support d'exécution. Ainsi, le modèle d'exécution d'une application est souvent éloigné de son code. Dans le premier, on a des *threads*, des tâches, des objets alors que dans le second on a des classes ou des interfaces. Les langages actuels semblent donc inadaptés pour manipuler des logiciels en exécution. Avec des collègues [6], nous avons commencé des reflexions autour des abstractions qu'un tel langage devrait contenir dans un cadre fonctionnel et leur sémantique. Ainsi, un tel langage doit :

1. être spécifique au domaine du contrôle de logiciel,
2. inclure des abstractions modernes comme le filtrage de motif, les compréhensions, des constructions de coordination, des automates, ...
3. être vérifié et donc reposé sur un système de type sophistiqué reposant sur des contrats de contrôle riches mais aussi sur les types sessions [P62] et la notion de *typestate* [P106],
4. avoir un support d'exécution capable de contrôler l'exécution de composant fonctionnel écrit dans d'autres langages,
5. disposer d'une sémantique formelle permettant de prouver des propriétés de sûreté d'exécution des programmes de contrôle.

Ce langage permettrait de manipuler des composants fonctionnels pour les installer, les configurer, les instancier, les mettre à jour et plus largement contrôler leurs automates de cycle de vie. Par exemple, on pourrait écrire des programme de la forme suivante :

```
let pl = node at /* une url */ in
let mission = install /* un paquet */ on pl in
let missionInstance = new mission(/* paramètres */) in
...
```

Le typage inclurait donc la vérification du déploiement qui serait statique si c'est possible ou déféré à l'exécution sinon. La manipulation des composants fonctionnels reposerait sur des contrats riches, comme par exemple :

```
type date = ...
type picture = ...
function contract MISSION_F = { shoot : date -> picture }
```

6. Antoine Beugnard, Jérémy Buisson, David Chemouil et Julien Brunel.

```
control contract MISSION_LF = {
  constructor : LF(initial_state : state_off | state_on)
  lifecycle :
    state_off -switch_on-> state_on -switch_off-> state_off
    state_on -freeze-> frozen -resume-> state_on
    frozen -switch_off-> state_off
}
control contract MISSION_C = {
  constructor : MI(initial_mode : safe | waiting)
  modes :
    safe -wait-> waiting -to_shooting-> shooting -sl_err-> safe
    shooting -wait-> waiting -sl_err-> safe
  typestate :
    shoot when shooting
    freeze when not shooting
}
```

La description des automates de contrôle des éléments permet de garantir que les programmes de contrôle sont corrects. Dans l'exemple, deux abstractions apparaissent : les états et les modes. Le cycle de vie correspond à l'automate des états, changer d'état est une opération visible de l'extérieur, elle peut avoir un impact les composants voisins. Le mode est, lui, encapsulé dans le composant (son changement n'est pas visible directement). Ici, les autres composants ne peuvent savoir si l'équipement est dans le mode où il prend des photos. Seuls les programmes de contrôle peuvent manipuler le mode. Notons que l'on peut limiter l'accès à des fonctions à des modes (ou des états), ici, la fonction `shoot` n'est accessible que dans le mode `shooting`.

Enfin, il est possible de créer des architectures, de les composer et de les décomposer :

```
let x = ... in
let y = ... in /* deux instances */
let arch = (| x, y, x.p => y.q |) in ... /* ou <= ou <=> */
let arch2 = arch & (| x ... |) in
match a with (| // une architecture
   x, y // des noms
   when x#name == "logger" && x._ <=> y._ // des contraintes
   with linkedCompo = [y] // des liens
|)  & tail -> ...
```

Dans le filtrage de motif, il y quatre parties : des noms d'instances qui seront liés par le filtrage, des contraintes sur les éléments qui peuvent être liés à ces noms, des déclarations de résultats qui seront exportées et une éventuelle continuation d'architecture. Ainsi, ici, on va chercher dans l'architecture `a`, des couples d'instances reliées dont l'une à le nom `"logger"` ; la liste des second membres de tels couples est exportée sous le nom `linkedCompo`.

Ces travaux très préliminaires ne sont présentés ici qu'à titre d'illustration pour donner une idée de la direction que pourrait prendre mes objectifs dans ce domaine.

Bibliographie

Publications personnelles

Ma liste de publication suit. Elle est classée par catégorie.

La figure 7.4 représente sous forme de courbe mes publications par année. J'ai, à ce jour, quarante publications dont 19,5% sont des publications en revues ou des chapitres de livre, 44% sont des articles dans des conférences et 36,5% sont des articles de *workshops*[7]. J'ai publié principalement en anglais (à 78%) et mes publications en français ont été faites dans des événements internationaux. Enfin, sur mes treize articles dans des conférences, il y une conférence A* (ICSE), 2 conférences A (ICFP et ICSSP), 3 conférences B (CBSE, SAC et FMOOD), 4 conférences C (ICSEA, FIT, SERA et DepCos) et 3 non classées.

J'ai eu 42 co-auteurs différents dont 5 sont des étudiants que j'ai encadré. Quatre articles sont conçus par une large équipe d'auteurs (plus de dix auteurs), soit 9,7% de mes articles. Sur les 37 autres articles, la moyenne de co-auteurs est de 3,1 auteurs par articles. Le maximum étant à deux auteurs pour 40% de mes articles.

Chapitre de livre

[1] Alexandre Cortier, Loïc Besnard, Jean-Paul Bodeveix, Fabien Dagnat, Gérald Garcia, Marc Pantel, Martin Strecker, Jean-Pierre Talpin, Ana-Elena Rugina, Julien Ouy, Mamoun Filali et Jérémy Buisson. « Synoptic : a domain-specific

7. Ici, il est important de souligner que tous les *workshops* dans lesquels j'ai publié avaient des processus de relecture et de selection des papiers.

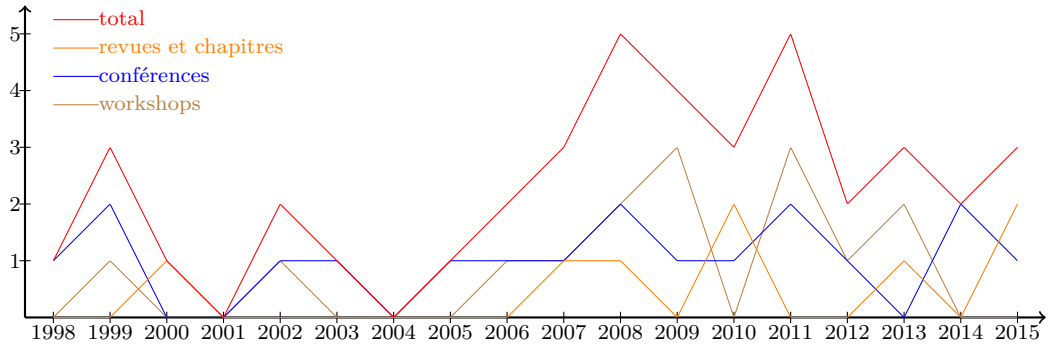


Figure 7.4 – Nombre de publications par an

modeling language for space on-board application software ». In : Engineering. Springer, 2010. Chap. Synthesis of embedded software, frameworks and methodologies for correctness by construction, p. 79–119. ISBN : 978-1-4419-6399-4.

Articles de journal international

[2] Jérémy Buisson, Fabien Dagnat, Elena Leroux et Sébastien Martinez. « Safe reconfiguration of Coqcots and Pycots components ». In : *Journal of Systems and Software* (2015). DOI : 10.1016/j.jss.2015.11.039.

[3] Loïc Besnard, Thierry Gautier, Julien Ouy, Jean-Pierre Talpin, Jean-Paul Bodeveix, Alexandre Cortier, Marc Pantel, Martin Strecker, Gérald Garcia, Ana-Elena Rugina, Jérémy Buisson et Fabien Dagnat. « Polychronous interpretation of synoptic, a domain specific modeling language for embedded flight-software. » In : *Formal Methods for Aerospace* 18 (2010), p. 80–87.

[4] Meriem Belguidoum et Fabien Dagnat. « Formalization of Component Substitutability ». In : *Electronic Notes in Theoretical Computer Science* 215 (juin 2008), p. 75–92. ISSN : 1571-0661. DOI : 10.1016/j.entcs.2008.06.022. URL : http://dx.doi.org/10.1016/j.entcs.2008.06.022.

[5] Meriem Belguidoum et Fabien Dagnat. « Dependency Management in Software Component Deployment ». In : *Electronic Notes in Theoretical Computer Science* 182 (juin 2007), p. 17–32. ISSN : 1571-0661. DOI : 10.1016/j.entcs.2006.09.029. URL : http://dx.doi.org/10.1016/j.entcs.2006.09.029.

[6] Fabien Dagnat, Marc Pantel, Matthias Colin et Patrick Sallé. « Typing Concurrent Objects and Actors ». In : *L'Objet – Méthodes formelles pour les objets* Vol 6.1/2000 (mai 2000), p. 83–106.

Articles de journal francophone

[7] Antoine Beugnard, Fabien Dagnat, Sylvain Guerin et Christophe Guychard. « Des situations de modélisation pour décrire un processus de modélisation ». In : *Ingénierie des Systèmes d'Information* 20.2 (2015), p. 41–66.

[8] Ali Koudri, Christophe Guychard, Sylvain Guérin, Fabien Dagnat, Antoine Beugnard et Joël Champeau. « De la nécessité de fédérer des modèles dans une chaîne d'outils ». In : *Génie logiciel : le magazine de l'ingénierie du logiciel et des systèmes* n°105 (juin 2013), p. 18–23.

Articles de conférence internationale

[9] Sébastien Martinez, Fabien Dagnat et Jérémy Buisson. « Pymoult : on-line updates for Python programs ». In : *Proceedings of the 10th International Conference on Software Engineering Advances (ICSEA'15)*. Barcelona, Spain, nov. 2015.

[10] Jérémy Buisson, Everton Calvacante, Fabien Dagnat, Sébastien Martinez et Elena Leroux. « Coqcots & Pycots : non-stopping components for safe dynamic reconfiguration ». In : *Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE'14)*. Lille, France, juin 2014, p. 85–90.

[11] Fahad Rafique Golra et Fabien Dagnat. « Generation of dynamic process models for multi-metamodel applications ». In : *Proceedings of the 8th International Conference on Software and System Process (ICSSP'12)*. Zurich, Switzerland, juin 2012, p. 48–57.

[12] Fahad Rafique Golra et Fabien Dagnat. « The lazy initialization multilayered modeling framework : NIER track ». In : *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. Honolulu, Hawaii, USA, mai 2011, p. 924–927.

[13] Fahad Rafique Golra et Fabien Dagnat. « Using component-oriented process models for multi-metamodel applications ». In : *Proceedings of the 9th IEEE International Conference on Frontiers of Information Technology (FIT'11)*. Islamabad, Pakistan, déc. 2011, p. 218–233.

[14] Jérémy Buisson et Fabien Dagnat. « ReCaml : execution state as the cornerstone of reconfigurations ». In : *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*. Baltimore, Maryland, USA, sept. 2010, p. 27–38.

[15] Jérémy Buisson et Fabien Dagnat. « Experiments with fractal on modular reflection ». In : *Proceedings of the 6th International Conference on Software Engineering Research, management and Applications (SERA'08)*. Prague, Czech Republic, août 2008, p. 179–186.

[16] David Chemouil et The SPaCIFY consortium. « The SPaCIFY project ». In : *Proceedings of the 12th International Space System Engineering Conference (DASIA'08)*. Palma de Majorca, Spain, mai 2008.

[17] Meriem Belguidoum et Fabien Dagnat. « Dependability in software component deployment ». In : *Proceedings of the 2nd IEEE International Conference on Dependability of Computer Systems (DepCos-RELCOMEX'07)*. Szklarska Poreba, Pologne, juin 2007, p. 223–230.

[18] Meriem Belguidoum et Fabien Dagnat. « Analysis of deployment dependencies in software components ». In : *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06)*. Dijon, France, avr. 2006.

[19] Laura Barrero Sastre, Fabien Dagnat, Emmanuel Donin De Rosiere, Nicolas Torneri et Ronan Keryell. « Bibtex++ : toward higher-order bibtexing ». In : *Proceedings of the 14th European TeX Conference (EuroTeX'03)*. Brest, France, juin 2003.

[20] Fabien Dagnat et Marc Pantel. « Static analysis of communications for Erlang ». In : *Proceedings of the 8th International Erlang User Conference (EUC'02)*. Stockholm, Sweden, nov. 2002.

[21] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat et Patrick Sallé. « Static safety analysis for non-uniform service availability in Actors ». In : *Proceedings of the 3rd International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*. Florence, Italy, fév. 1999.

Articles de conférence francophone

[22] Antoine Beugnard, Fabien Dagnat, Sylvain Guerin et Christophe Guychard. « Des situations de modélisation pour évaluer les outils de modélisation ». In : *Actes du 32ème congrès de l'INFormatique des ORganisations et Systèmes d'Information et de Décision (INFORSID'14)*. 2014, p. 181–196.

[23] Meriem Belguidoum et Fabien Dagnat. « Vers un déploiement sûr et flexible des composants logiciels ». In : *Actes de la 9ième Conférence Internationale sur les Nouvelles Technologies de la Répartition (NOTERE'09)*. Montréal, Canada, juin 2009.

[24] Meriem Belguidoum, Fabien Dagnat et Antoine Beugnard. « Analyse des dépendances pour le déploiement automatique de composants ». In : *Actes de la 4ème Conférence Francophone autour des Composants Logiciels (JC'05)*. Le Croisic, France, avr. 2005, p. 57–68.

[25] Matthias Colin, Marc Pantel, Fabien Dagnat et Patrick Sallé. « Intégration des Typages Fonctionnel et Concurrent d'un Langage Fonctionnel d'Acteurs ». In : *Actes des 10ième Journées Francophones des Langages Applicatifs (JFLA'99)*. Avoriaz, France, fév. 1999.

[26] Fabien Dagnat, Marc Pantel et Patrick Sallé. « ML-Act, un langage fonctionnel d'Acteurs ». In : *Actes des 9ième Journées Francophones des Langages Applicatifs (JFLA'98)*. Lac de Côme, Italie, fév. 1998.

Articles de *workshop* international

[27] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard et Fabien Dagnat. « Conceptual interoperability through Models Federation ». In : *Proceedings of the Semantic Information Federation Community Workshop (SIMF'13)*. Miami, United States, oct. 2013. URL : https://hal.archives-ouvertes.fr/hal-00905036.

[28] Sebastien Martinez, Fabien Dagnat et Jérémy Buisson. « Prototyping DSU Techniques Using Python ». In : *Proceedings of the 5th Workshop on Hot Topics in Software Upgrades (HotSWUp'13)*. San José, California, USA, juin 2013.

[29] Fahad Rafique Golra et Fabien Dagnat. « Specifying the Interaction Control Behavior of a Process Model using Hierarchical Petri Net ». In : *Proceedings of the 2nd Workshop on Process-based approaches for Model-Driven Engineering (PMDE'12)*. Lyngby, Denmark, juil. 2012.

[30] Sylvain Bouveret, Julien Brunel, David Chemouil et Fabien Dagnat. « Towards a categorical framework to ensure correct software evolutions ». In : *Proceedings of the 3rd Workshop on Hot Topics in Software Upgrades (HotSWUp'11)*. Hannover, Germany, avr. 2011.

[31] Fabien Dagnat, Gwendal Simon et Xu Zhang. « Toward a distributed package management system ». In : *Proceedings of the 2nd Workshop on logics for component configuration (Lococo'11)*. Perugia, Italy, sept. 2011.

[32] Fahad Rafique Golra et Fabien Dagnat. « Component-oriented multi-metamodel process modeling framework (CoMProM) ». In : *Proceedings of the 1st Workshop on Process-based approaches for Model-Driven Engineering (PMDE'11)*. Birmingham, UK, juil. 2011, p. 50–59.

[33] Meriem Belguidoum et Fabien Dagnat. « Integrating extra-functional properties in component deployment dependencies ». In : *Proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS'09)*. Eindhoven, the Netherlands, nov. 2009, p. 195–209.

[34] Antoine Beugnard, Sophie Chabridon, Denis Conan, Fabien Dagnat, Eveline Kaboré et Chantal Taconet. « Towards context-aware components ». In : *Foundations of Software Engineering : Proceedings of the 1st International Workshop on Context-Aware Software Technology and Applications (CASTA'09)*. Amsterdam, The Netherlands, août 2009, p. 1–4.

[35] Alexandre Cortier, Loïc Besnard, Jean-Paul Bodeveix, Jérémy Buisson, Fabien Dagnat, Mamoun Filali, Gérald Garcia, Thierry Gautier, Julien Ouy, Marc Pantel, Ana-Elena Rugina, Martin Strecker et Jean-Pierre Talpin. « Synoptic : a domain specific modeling language for embedded flight-software ». In : *Proceedings of the 1st Workshop on Formal Methods for Aerospace (FMA'09)*. Eindhoven, the Netherlands, nov. 2009.

[36] Jérémy Buisson, Cecilia Carro et Fabien Dagnat. « Issues in applying a model driven approach to reconfigurations of satellite software ». In : *Proceedings of the 1st Workshop on Hot Topics in Software Upgrades (HotSWUp'08), Nashville, Tennessee, USA*. Oct. 2008.

[37] Jérémy Buisson et Fabien Dagnat. « Introspecting continuations in order to update active code ». In : *Proceedings of the 1st Workshop on Hot Topics in Software Upgrades (HotSWUp'08), Nashville, Tennessee, USA*. Oct. 2008.

[38] Meriem Belguidoum et Fabien Dagnat. « Formalization of component substitutability ». In : *Proceedings ot the 4th International Workshop on Formal Aspects of Component Software (FACS'07)*. Sophia-Antipolis, France, sept. 2007.

[39] Meriem Belguidoum et Fabien Dagnat. « Dependency management in software component deployment ». In : *Proceedings of the 3rd International Workshop on Formal Aspects of Component Software (FACS'06)*. Prague, Czech republic, sept. 2006.

Articles de *workshop* francophone

[40] Fabien Dagnat, Marc Pantel et Patrick Sallé. « Vers un erlang Typé ? » In : *Actes des 9ième Journées Formalisation d'Activités Concurrentes*. Fév. 2002.

[41] Matthias Colin, Marc Pantel, Fabien Dagnat et Patrick Sallé. « Analyses statiques fonctionnelles et concurrentes sur un langage d'acteurs ». In : *Actes des 6ième Journées Formalisation d'Activités Concurrentes*. Fév. 1999.

Autres publications

[42] Antoine Beugnard, Fabien Dagnat et Eveline Kaboré. *Processus de conception d'une application sensible au contexte*. Rapp. tech. Télécom Bretagne, 2009.

[43] Fabien Dagnat. « Vérification Statique de Programmes Répartis ». Thèse de doctorat. Institut National Polytechnique de Toulouse, mai 2001.

[44] Fabien Dagnat. « Conception d'un Langage Fonctionnel d'Acteur et Réalisation de son Compilateur ». DEA IFP. Institut National Polytechnique de Toulouse, sept. 1997.

Publications d'autres auteurs

[P1] Wil M. P. van der Aalst. « Formalization and verification of event-driven process chains ». In : *Information and Software Technology* 41.10 (1999), p. 639–650. URL : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.1015.

[P2] Wil M. P. van der Aalst, Hajo A Reijers, Anton JMM Weijters, Boudewijn F van Dongen, AK Alves de Medeiros, Minseok Song et HMW Verbeek. « Business process mining : An industrial application ». In : *Information Systems* 32.5 (2007), p. 713–732. ISSN : 0306-4379. DOI : 10.1016/j.is.2006.05.003.

[P3] Pietro Abate, Roberto DiCosmo, Ralf Treinen et Stefano Zacchiroli. « MPM : A Modular Package Manager ». In : *Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering*. CBSE '11. Boulder, Colorado, USA : ACM, 2011, p. 179–188. ISBN : 978-1-4503-0723-9. DOI : 10.1145/2000229.2000255.

[P4] Michael J. Adams, Arthur H. M. ter Hofstede, David Edmond et Wil M. P. van der Aalst. « Facilitating Flexibility and Dynamic Exception Handling in Workflows through Worklets ». In : *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05) Forum*. Sous la dir. d'Orlando Bello, Johann Eder, Oscar Pastor et Joao Falcao e Cunha. Porto, Portugal : FEUP Edicoes, 2005, p. 45–50. URL : http://www.ceur-ws.org/Vol-161/FORUM_08.pdf.

[P5] Jonathan Aldrich. « Using Types to Enforce Architectural Structure ». In : *WICSA '08 : Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*. Washington, DC, USA : IEEE Computer Society, 2008, p. 211–220. ISBN : 978-0-7695-3092-5. DOI : 10.1109/WICSA.2008.48.

[P6] Jonathan Aldrich, Craig Chambers et David Notkin. « ArchJava : connecting software architecture to implementation ». In : *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*. Orlando, Florida : ACM, 2002, p. 187–197. ISBN : 1-58113-472-X. DOI : 10.1145/581339.581365.

[P7] Robert Allen et David Garlan. « A formal basis for architectural connection ». In : *ACM Transactions on Software Engineering and Methodology* 6.3 (1997), p. 213–249. ISSN : 1049-331X. DOI : 10.1145/258077.258078.

[P8] Gautam Altekar, Ilya Bagrak, Paul Burstein et Andrew Schultz. « OPUS : online patches and updates for security ». In : *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. Baltimore, Maryland, USA, août 2005, p. 287–302. URL : http://dl.acm.org/citation.cfm?id=1251398.1251417.

[P9] Emmanuelle Anceaume, Maria Gradinariu, Ajoy Kumar Datta, Gwendal Simon et Antonino Virgillito. « A Semantic Overlay for Self- Peer-to-Peer Publish/Subscribe ». In : *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), 4-7 July 2006, Lisboa, Portugal*. IEEE Computer Society, 2006, p. 22. ISBN : 0-7695-2540-7. DOI : 10.1109/ICDCS.2006.12.

[P10] Jonathan Appavoo, Kevin Hui, Craig Soules, Robert Wisniewski, Dilma Da Silva, Orran Krieger, David Edelsohn, Marc Auslander, Ben Gamsa, Gregory Ganger, Paul McKenney, Michal Ostrowski, Bryan Rosenburg, Michael Stumm et Jimi Xenidis. « Enabling autonomic behavior in systems software with hot-swapping ». In : *IBM Systems Journal* 42.1 (jan. 2003). doi : 10.1147/sj.421.0060.

[P11] Joe Armstrong. *Programming Erlang : software for a concurrent world*. The Pragmatic PRogrammers, 2013. isbn : 978-1-93778-553-6. url : https://pragprog.com/book/jaerlang2/programming-erlang.

[P12] Jeff Arnold et M. Frans Kaashoek. « Ksplice : automatic rebootless kernel updates ». In : *European Conference on Computer Systems*. Nuremberg, Germany, avr. 2009, p. 187–198. doi : 10.1145/1519065.1519085.

[P13] Dusan Bálek et Frantisek Plasil. « Software Connectors and Their Role in Component Deployment ». In : *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems*. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 2001, p. 69–84. isbn : 0-7923-7481-9. url : http://dl.acm.org/citation.cfm?id=648289.754241.

[P14] John J. Barton. « Software Upgrade in Ubiquitous Computing. » In : *Pervasive 2004, Vienna, Austria*. Avr. 2004.

[P15] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio et Christian Pérez. « GCM : a grid extension to Fractal for autonomous distributed components ». In : *Annals of Telecommunications* 64 (1 2009), p. 5–24. issn : 0003-4347. doi : 10.1007/s12243-008-0068-8.

[P16] Andrew Baumann, Jonathan Appavoo, Robert Wisniewski, Dilma Da Silva, Orran Krieger et Gernot Heiser. « Reboots Are for Hardware : Challenges and Solutions to Updating an Operating System on the Fly ». In : *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA : USENIX Association, juin 2007. url : http://dl.acm.org/citation.cfm?id=1364385.1364411.

[P17] Meriem Belguidoum. « Conception d'une infrastructure pour un déploiement sûr et flexible des composants logiciels ». Thèse de doctorat. École Nationale Supérieure des Télécommunications de Bretagne, fév. 2008. url : https://www.telecom-bretagne.eu/publications/publication.php?idpublication=1445.

[P18] Reda Bendraou, Benoit Combemale, X. Cregut et M.-P. Gervais. « Definition of an Executable SPEM 2.0 ». In : *Proceedings of the 14th Asia-Pacific Software Engineering Conference, 2007. APSEC 2007*. Déc. 2007, p. 390–397. doi : 10.1109/ASPEC.2007.60.

[P19] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins. « Making Components Contract Aware ». In : *Computer* 32.7 (juil. 1999), p. 38–45. issn : 0018-9162. doi : 10.1109/2.774917.

[P20] Jean Bézivin, Frédéric Jouault et Patrick Valduriez. « On the Need for Megamodels ». In : *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004. url : http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OOPSLA04/bezivin-megamodel.pdf.

[P21] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle et Keith Wansbrough. *Dynamic rebinding for marshalling and update, with destruct-time λ*. Rapp. tech. UCAM-CL-TR-568. University of Cambridge, Computer Laboratory, fév. 2004. URL : http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-568.pdf.

[P22] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid et Yihui Tang. « On the False-positive Rate of Bloom Filters ». In : *Inf. Process. Lett.* 108.4 (oct. 2008), p. 210–213. ISSN : 0020-0190. DOI : 10.1016/j.ipl.2008.05.018.

[P23] Premek Brada. « Specification-Based Component Substitutability and Revision Identification ». Thèse de doct. Prague : Charles University, août 2003. URL : http://d3s.mff.cuni.cz/publications/download/brada_phd.pdf.

[P24] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma et Jean-Bernard Stefani. « The Fractal component model and its support in Java ». In : *Software : Practice and Experience* 36.11-12 (sept. 2006). Spec. issue on experiences with auto-adaptive and reconfigurable systems, p. 1257–1284. ISSN : 0038-0644. DOI : 10.1002/spe.767.

[P25] Tomas Bures, Petr Hnetynka et Frantisek Plasil. « SOFA 2.0 : Balancing Advanced Features in a Hierarchical Component Model. » In : *SERA*. IEEE Computer Society, 2006, p. 40–48. ISBN : 0-7695-2656-X. DOI : 10.1109/SERA.2006.62.

[P26] Antonio Carzaniga, Alfonso Fuggetta, Richard S. Hall, Dennis Heimbigner et Alexander L. Wolf. *A Characterization Framework for Software Deployment Technologies*. Rapp. tech. CU-CS-857-98. Department of Computer Science, University of Colorado, avr. 1998. URL : http://scholar.colorado.edu/csci_techreports/806.

[P27] John Cheesman et John Daniels. *UML Components : A Simple Process for Specifying Component-Based Software*. Addison Wesley Professional, 2001. ISBN : 0-201-70851-5.

[P28] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang et Pen-Chung Yew. « POLUS : A POwerful Live Updating System ». In : *Proceedings of the 29th International Conference on Software Engineering*. ICSE '07. Washington, DC, USA : IEEE Computer Society, 2007, p. 271–281. ISBN : 0-7695-2828-7. DOI : 10.1109/ICSE.2007.65.

[P29] Ken Christensen, Allen Roginsky et Miguel Jimeno. « A New Analysis of the False Positive Rate of a Bloom Filter ». In : *Inf. Process. Lett.* 110.21 (oct. 2010), p. 944–949. ISSN : 0020-0190. DOI : 10.1016/j.ipl.2010.07.024.

[P30] Antonio Cicchetti, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio et Stefano Zacchiroli. « A Model Driven Approach to Upgrade Package-Based Software Systems ». English. In : *Evaluation of Novel Approaches to Software Engineering*. Sous la dir. de LeszekA. Maciaszek, César González-Pérez et Stefan Jablonski. T. 69. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2010, p. 262–276. ISBN : 978-3-642-14818-7. DOI : 10.1007/978-3-642-14819-4_19.

[P31] Denis Conan, Romain Rouvoy et Lionel Seinturier. « COSMOS : composition de noeuds de contexte ». In : *Revue Technique et Science Informatiques (TSI)* 27.9-10 (2008), p. 1189–1224. URL : http://hal.inria.fr/inria-00330574.

[P32] Geoff Coulson, Gordon Blair, Paul Grace, Francosi Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama et Thirunavukkarasu Sivaharan. « A generic component model for building systems software ». In : *ACM Transactions on Computer Systems* 26.1 (fév. 2008). doi : 10.1145/1328671.1328672.

[P33] Alan Dearle, Graham N. C. Kirby et Andrew J. McCarthy. « A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications ». In : *Inter. Conf. on Autonomic Computing*. Washington, DC, USA : IEEE, 2004, p. 300–301. isbn : 0-7695-2114-2. url : http://portal.acm.org/citation.cfm?id=1078026.1078439.

[P34] Mikhail Dmitriev. « Safe class and data evolution in large and long-lived Java™ applications ». published as Sun Microsystems TR-2001-98. Thèse de doct. University of Glasgow, mar. 2001. url : http://dl.acm.org/citation.cfm?id=975002.

[P35] Eelco Dolstra, Andres Löh et Nicolas Pierron. « NixOS : A purely functional Linux distribution ». In : *Journal of Functional Programming* 20 (Special Issue 5-6 2010), p. 577–615. issn : 1469-7653. doi : 10.1017/S0956796810000195.

[P36] Kent Dybvig, Simon Peyton-Jones et Amr Sabry. « A monadic framework for delimited continuations ». In : *Journal of Functional Programming* 17.6 (nov. 2007), p. 687–730. doi : 10.1017/S0956796807006259.

[P37] Mturi Elias et Paul Johannesson. « A Survey of Process Model Reuse Repositories ». In : *Information Systems, Technology and Management : 6th International Conference, ICISTM 2012, Grenoble, France, March 28-30, 2012. Proceedings*. Sous la dir. de Sumeet Dua, Aryya Gangopadhyay, Parimala Thulasiraman, Umberto Straccia, Michael Shepherd et Benno Stein. T. 285. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, p. 64–76. doi : 10.1007/978-3-642-29166-1_6.

[P38] Jean-Marie Favre et Tam NGuyen. « Towards a Megamodel to Model Software Evolution through Transformations ». In : *Electronic Notes in Theoretical Computer Science* 127.3 (2005), p. 59–74. issn : 1571-0661. doi : 10.1016/j.entcs.2004.08.034.

[P39] Peter H. Feiler et David P. Gluch. *Model-Based Engineering with AADL : An Introduction to the SAE Architecture Analysis & Design Language*. 1st. Addison-Wesley Professional, 2012. isbn : 9780321888945. url : http://www.pearsonhighered.com/educator/product/ModelBased-Engineering-with-AADL-An-Introduction-to-the-SAE-Architecture-Analysis-Design-Language/9780321888945.page.

[P40] Matthias Felleisen. « The theory and practice of first-class prompts ». In : *Principles of Programming Languages*. San Diego, California, USA, jan. 1988, p. 180–190. doi : 10.1145/73560.73576.

[P41] Areski Flissi, Jérémy Dubus, Nicolas Dolet et Philippe Merle. « Deploying on the Grid with DeployWare ». In : *Inter. Symp. on Cluster Computing and the Grid*. IEEE, mai 2008, p. 177–184. url : http://hal.inria.fr/hal-00259836.

[P42] Alfonso Fuggetta. « Software process : a roadmap ». In : *Proceedings of the Conference on The Future of Software Engineering*. ICSE'00. Limerick, Ireland : ACM, 2000, p. 25–34. isbn : 1-58113-253-0. doi : 10.1145/336512.336521.

[P43] David Garlan, Robert T. Monroe et David Wile. « ACME : Architectural Description of Component-based Systems ». In : *Foundations of Component-based Systems*. Sous la dir. de Gary T. Leavens et Murali Sitaraman. New York, NY, USA : Cambridge University Press, 2000, p. 47–67. isbn : 0-521-77164-1. url : http://dl.acm.org/citation.cfm?id=336431.336437.

[P44] Thomas Genssler, Alexander Christoph, Michael Winter, Oscar Nierstrasz, Stéphane Ducasse, Roel Wuyts, Gabriela Arévalo, Bastiaan Schönhage, Peter Müller et Chris Stich. « Components for embedded software : the PECOS approach ». In : *CASES '02 : Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. Grenoble, France : ACM, 2002, p. 19–26. isbn : 1-58113-575-0. doi : 10.1145/581630.581634.

[P45] Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi et Hassan Haghighi. « An Architectural Approach to Ensure Globally Consistent Dynamic Reconfiguration of Component-based Systems ». In : *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*. CBSE '12. Bertinoro, Italy : ACM, 2012, p. 177–182. isbn : 978-1-4503-1345-2. doi : 10.1145/2304736.2304765.

[P46] Stephen Gilmore, Dilsun Kírlí et Chris Walton. *Dynamic ML without dynamic types*. Rapp. tech. ECS-LFCS-97-378. The University of Edinburgh, 1997. url : http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.939.

[P47] Cristiano Giuffrida, Anton Kuijsten et Andrew S. Tanenbaum. « Safe and Automatic Live Update for Operating Systems ». In : *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA : ACM, 2013, p. 279–292. isbn : 978-1-4503-1870-9. doi : 10.1145/2451116.2451147.

[P48] Yann Glouche, Paul Le Guernic, Jean-Pierre Talpin et Thierry Gautier. *A Boolean algebra of contracts for logical assume-guarantee reasoning*. Anglais. Rapport de recherche RR-6570. INRIA, 2008, p. 41. url : http://hal.inria.fr/inria-00292870.

[P49] Fahad Rafique GOLRA. « A Refinement based methodology for software process modeling ». Theses. Télécom Bretagne, Université de Rennes 1, jan. 2014. url : https://tel.archives-ouvertes.fr/tel-00978732.

[P50] Object Management Group. *Unified Modeling Language Specification, version 2.5*. Juin 2015. url : http://www.omg.org/spec/UML/2.5.

[P51] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang et Jian Lu. « Javelus : A Low Disruptive Approach to Dynamic Software Updates. » In : *APSEC*. Sous la dir. de Karl R. P. H. Leung et Pornsiri Muenchaisri. IEEE, 2012, p. 527–536. isbn : 978-0-7695-4922-4. doi : 10.1109/APSEC.2012.55. url : http://dblp.uni-trier.de/db/conf/apsec/apsec2012.html#GuCXMZL12.

[P52] Carl A. Gunter, Didier Rémy et Jon G. Riecke. « A generalization of exceptions and control in ML-like languages ». In : *International Conference on Functional Programming Languages and Computer Architecture*. La Jolla, California, USA, juin 1995, p. 12–23. doi : 10.1145/224164.224173.

[P53] Deepak Gupta et Pankaj Jalote. « On-line software version change using state transfer between processes ». In : *Software : Practice and Experience* 23.9 (sept. 1993), p. 949–964. doi : 10.1002/spe.4380230903.

[P54] Deepak Gupta, Pankaj Jalote et Gautam Barua. « A formal framework for on-line software version change ». In : *IEEE Transactions on Software Engineering* 22.2 (fév. 1996), p. 120–131. doi : 10.1109/32.485222.

[P56] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks et Jeffrey S. Foster. « Kitsune : efficient, general-purpose dynamic software updating for C. » In : *OOPSLA*. Sous la dir. de Gary T. Leavens et Matthew B. Dwyer. ACM, 2012, p. 249–264. isbn : 978-1-4503-1561-6. doi : 10.1145/2384616.2384635. url : http://dblp.uni-trier.de/db/conf/oopsla/oopsla2012.html#HaydenSDHF12.

[P55] Christopher M. Hayden, Edward K. Smith, Michael Hicks et Jeffrey S. Foster. « State transfer for clear and efficient runtime upgrades ». In : 2011, p. 179–184. isbn : 978-1-4244-9195-7. doi : 10.1109/ICDEW.2011.5767632.

[P57] Patrick J. Hayes. *The frame problem and related problems in Artificial Intelligence*. Rapp. tech. Stanford, CA, USA : Stanford University, 1971.

[P58] Michael Hicks et Scott Nettles. « Dynamic software updating ». In : *ACM Transactions on Programming Languages and Systems* 27.6 (nov. 2005), p. 1049–1096. doi : 10.1145/1108970.1108971.

[P59] Didier Hoareau et Yves Mahéo. « Constraint-Based Deployment of Distributed Components in a Dynamic Network ». In : *Architecture of Computing Systems - ARCS 2006 : 19th International Conference, Frankfurt/Main, Germany, March 13-16, 2006. Proceedings*. Sous la dir. de Werner Grass, Bernhard Sick et Klaus Waldschmidt. LNCS. Springer Berlin Heidelberg, mar. 2006, p. 450–464. isbn : 978-3-540-32766-0. doi : 10.1007/11682127_32.

[P60] Didier Hoareau et Yves Mahéo. « Middleware support for the deployment of ubiquitous software components ». In : *Personal and Ubiquitous Computing* 12 (2 fév. 2008), p. 167–178. issn : 1617-4909. doi : 10.1007/s00779-006-0110-7.

[P61] David Hollingsworth. « The Workflow Reference Model : 10 Years On ». In : *Technical Committee Chair of WfMC*. 2004, p. 295–312.

[P62] Kohei Honda, Vasco Thudichum Vasconcelos et Makoto Kubo. « Language Primitives and Type Discipline for Structured Communication-Based Programming ». In : *Proceedings of the 7th European Symposium on Programming : Programming Languages and Systems*. ESOP '98. London, UK, UK : Springer-Verlag, 1998, p. 122–138. isbn : 3-540-64302-8. url : http://dl.acm.org/citation.cfm?id=645392.651876.

[P63] Frédéric Jouault, Jean Bézivin et Ivan Kurtev. « TCS : a DSL for the specification of textual concrete syntaxes in model engineering ». In : *Proceedings of the 5th international conference on Generative programming and component engineering*. GPCE '06. Portland, Oregon, USA : ACM, 2006, p. 249–254. isbn : 1-59593-237-2. doi : 10.1145/1173706.1173744.

[P64] Stuart Kent. « Model Driven Engineering ». In : *Integrated Formal Methods*. Sous la dir. de Michael Butler, Luigia Petre et Kaisa Sere. T. 2335. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, p. 286–298. isbn : 978-3-540-43703-1. doi : 10.1007/3-540-47884-1_16.

[P65] Jeffrey O. Kephart et David M. Chess. « The vision of autonomic computing ». In : *Computer* 36.1 (2003), p. 41–50. ISSN : 0018-9162. DOI : 10.1109/MC.2003.1160055.

[P66] Jeff Kramer et Jeff Magee. « The evolving philosophers problem : dynamic change management ». In : *IEEE Transactions on Software Engineering* 16.11 (nov. 1990), p. 1293–1306. DOI : 10.1109/32.60317.

[P67] Jeff Kramer et Jeff Magee. « The evolving philosophers problem : dynamic change management ». In : *IEEE Transaction on Software Engineering* 16.11 (nov. 1990), p. 1293–1306. DOI : 10.1109/32.60317.

[P68] Kung-Kiu Lau, Perla Velasco Elizondo et Zheng Wang. « Exogenous Connectors for Software Components ». In : *Proceedings of the 8th International Conference on Component-Based Software Engineering, CBSE, St. Louis, Mo, USA*. Springer-Verlag, mai 2005, p. 90–106. ISBN : 978-3-540-25877-3. DOI : 10.1007/11424529_7.

[P69] Kung-Kiu Lau et Zheng Wang. « Software Component Models ». In : *IEEE Transactions on Software Engineering* 33.10 (2007), p. 709–724. ISSN : 0098-5589. DOI : 10.1109/TSE.2007.70726.

[P70] Daniel Le Berre et Pascal Rapicault. « Dependency management for the Eclipse ecosystem ». In : *Proceedings of the 1st International Workshop on Open Component Ecosystems (IWOCE'09)*. Amsterdam, The Netherlands : ACM, août 2009, p. 21–30. ISBN : 978-1-60558-677-9. DOI : 10.1145/1595800.1595805.

[P71] Daniel Le Berre et Stéphanie Roussel. « Sat4j 2.3.2 : on the fly solver configuration, System Description ». In : *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)* (2013). URL : https://satassociation.org/jsat/index.php/jsat/article/view/109.

[P72] Xavier Leroy. *The ZINC experiment, an economical implementation of the ML language*. Rapp. tech. 117. INRIA, 1990. URL : http://caml.inria.fr/pub/papers/xleroy-zinc.pdf.

[P73] Barbara H. Liskov et Jeannette M. Wing. *Behavioral subtyping using invariants and constraints*. MU CS-99-156. School of Computer Science, Carnegie Mellon University, juil. 1999.

[P74] Kristis Makris et Rida A. Bazzi. « Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction ». In : *Proceedings of the USENIX '09 Annual Technical Conference*. USENIX Association, juin 2009. URL : https://www.usenix.org/legacy/event/usenix09/tech/full_papers/makris/makris.pdf.

[P75] Abdul Malik Khan, Sophie Chabridon et Antoine Beugnard. « A dynamic approach to consistency management for mobile multiplayer games ». In : *Inter. Conf. on New technologies in distributed systems*. Lyon, France : ACM, 2008, 42 :1–42 :6. ISBN : 978-1-59593-937-1. DOI : 10.1145/1416729.1416783.

[P76] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy et Ralf Treinen. « Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. » In : *ASE*. IEEE Computer Society, 2006, p. 199–208. ISBN : 0-7695-2579-2. DOI : 10.1109/ASE.2006.49.

[P77] Selma MATOUGUI. « Proposition d'un processus de réification d'abstraction de communication comme un connecteur associé à des générateurs ». Thèse de doctorat. Université de Rennes 1, déc. 2005. URL : https://tel.archives-ouvertes.fr/tel-00012036/.

[P78] Selma MATOUGUI et Antoine BEUGNARD. « Two Ways of Implementing Software Connections Among Distributed Components ». In : *On the Move to Meaningful Internet Systems 2005 : CoopIS, DOA, and ODBASE : OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings Part II*. Sous la dir. de Robert MEERSMAN et Zahir TARI. T. 3761. LNCS. Springer Berlin Heidelberg, 2005, p. 997–1014. ISBN : 978-3-540-32120-0. DOI : 10.1007/11575801_5.

[P79] Nikunj R. MEHTA, Nenad MEDVIDOVIC et Sandeep PHADKE. « Towards a Taxonomy of Software Connectors ». In : *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland : ACM, 2000, p. 178–187. ISBN : 1-58113-206-9. DOI : 10.1145/337180.337201.

[P80] Bertrand MEYER. *Object-Oriented Software Construction (2nd ed.)* Prentice Hall, 1997.

[P81] Emili MIEDES et Francesc D. MUÑOZ-ESCOÍ. *A Survey about Dynamic Software Updating*. Rapp. tech. ITI-SIDI-2012/003. Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, mai 2012. URL : http://web.iti.upv.es/~fmunyoz/research/pdf/TR-ITI-SIDI-2012003.pdf.

[P82] Iulian NEAMTIU, Michael HICKS, Gareth STOYLE et Manuel ORIOL. « Practical dynamic software updating for C ». In : *Programming Language Design and Implementation*. Ottawa, Canada, juin 2006, p. 72–83. DOI : 10.1145/1133255.1133991.

[P83] OASIS. *Web Services Business Process Execution Language (WS-BPEL), Version 2.0*. Version 2.0. OASIS, mai 2007. URL : https://www.oasis-open.org/committees/download.php.

[P84] OMG. *Deployment and Configuration of Component-based Distributed Applications Specification, Version 4*. Avr. 2006. URL : http://www.omg.org/cgi-bin/doc?formal/06-04-02.

[P85] OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 3 : CORBA Component Model*. Jan. 2008. URL : http://www.omg.org/spec/CORBA/3.1/Components/PDF.

[P86] OMG. *Software & Systems Process Engineering Metamodel Specification (SPEM), Version 2.0*. Version 2.0. Object Management Group, Inc., avr. 2008. URL : http://www.omg.org/spec/SPEM/2.0/PDF.

[P87] OMG. *Business Process Model And Notation (BPMN), Version 2.0*. Version 2.0. Object Management Group, Inc., jan. 2011. URL : http://www.omg.org/spec/BPMN/2.0/PDF.

[P88] Rob van OMMERING, Frank van der LINDEN, Jeff KRAMER et Jeff MAGEE. « The Koala Component Model for Consumer Electronics Software ». In : *Computer* 33.3 (2000), p. 78–85. ISSN : 0018-9162. DOI : 10.1109/2.825699.

[P89] ORACLE. *Enterprise Java Beans, version 3.2*. 2013. URL : http://download.oracle.com/otndocs/jcp/ejb-3_2-fr-eval-spec/index.html.

[P90] OSGi. *OSGi services platform specification, Release3*. Open Services Gateway initiative. Mar. 2003. URL : http://www.osgi.org.

[P91] Dan Österberg et Johan Lilius. *Rethinking Software Updating : Concepts for Improved Updatability*. Rapp. tech. 550. Turku Centre for Computer Science, sept. 2003. URL : http://tucs.fi/publications/view/?pub_id=tOsJ03a.

[P92] Leon J. Osterweil. « Software processes are software too ». In : *Proceedings of the 9th international conference on Software Engineering*. ICSE '87. Monterey, California, USA : IEEE Computer Society Press, 1987, p. 2–13. ISBN : 0-89791-216-0. URL : http://dl.acm.org/citation.cfm?id=41765.41766.

[P93] Mourad Oussalah, éd. *Ingénierie des composants : concepts, techniques et outils*. Génie logiciel. Paris : Vuibert informatique, 2005. ISBN : 2-7117-4836-7.

[P94] Allen Parrish, Brandon Dixon et David Cordes. « A conceptual foundation for component-based software deployment ». In : *Journal of Systems and Software* 57.3 (2001), p. 193–200. ISSN : 0164-1212. DOI : 10.1016/S0164-1212(01)00009-7.

[P95] David Pescovitz. « Monsters in a box ». In : *Wired* 8.12 (déc. 2000), p. 341–347. URL : http://www.wired.com/wired/archive/8.12/supercomputers_pr.html.

[P96] Pierre-Yves Pillain, Joel Champeau et Hanh Nhi Tran. « Towards an Enactment Mechanism for MODAL Process Models ». In : *First Workshop on Process-based approaches for Model-Driven Engineering (PMDE), 2011*. Juin 2011, p. 33.

[P97] Luís Pina et Michael Hicks. « Rubah : Efficient, General-purpose Dynamic Software Updating for Java ». In : *Presented as part of the 5th Workshop on Hot Topics in Software Upgrades*. San Jose, CA : USENIX, 2013. URL : https://www.usenix.org/conference/hotswup13/workshop-program/presentation/Pina.

[P98] *Proceedings of the 1st Workshop on Hot Topics in Software Upgrades (HotSWUp'08), Nashville, Tennessee, USA*. Oct. 2008.

[P99] Ricardo Sanz, Manuel Rodriguez, Carlos Martínez et Adolfo Hernando. « Embedded Component Technology for Complex Control Systems ». In : *Proceedings of the 17th IFAC World Congress*. T. 17. IFAC. 2008, p. 6897–6902. DOI : 10.3182/20080706-5-KR-1001.01169.

[P100] João Costa Seco et Luís Caires. « A Basic Model of Typed Components ». In : *ECOOP '00 : Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 2000, p. 108–128. ISBN : 3-540-67660-0. DOI : 10.1007/3-540-45102-1_6.

[P101] Habib Seifzadeh, Hassan Abolhassani et Mohsen Sadighi Moshkenani. « A survey of dynamic software updating ». In : *Journal of Software : Evolution and Process* (2012). ISSN : 2047-7481. DOI : 10.1002/smr.1556.

[P102] Lionel Seinturier, Nicolas Pessemier, Laurence Duchien et Thierry Coupaye. « A Component Model Engineered with Components and Aspects ». In : *Component-Based Software Engineering*. Sous la dir. d'Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens Szyperski et Kurt Wallnau. T. 4063. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, p. 139–153. ISBN : 978-3-540-35628-8. DOI : 10.1007/11783565_10.

[P103] David N. Smith. *Dave's Smalltalk FAQ*. IBM. Juil. 1995. URL : http://ftp.sunet.se/pub/lang/smalltalk/faq/SmallFaq.html.

[P104] Gareth P. Stoyle. « A Theory of Dynamic Software Updates ». Thèse de doct. University of Cambridge, 2006. URL : http://www.cl.cam.ac.uk/%5C~%7B%7Dpes20/GarethStoyle%5C_thesis.pdf.

[P105] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell et Iulian Neamtiu. « Mutatis Mutandis : safe and predictable dynamic software updating ». In : *Annual Symposium on Principles of Programming Languages*. Long Beach, California, USA, jan. 2005, p. 183–194. DOI : 10.1145/1047659.1040321.

[P106] Robert E Strom et Shaula Yemini. « Typestate : A Programming Language Concept for Enhancing Software Reliability ». In : *IEEE Transaction on Software Engineering* 12.1 (jan. 1986), p. 157–171. ISSN : 0098-5589. DOI : 10.1109/TSE.1986.6312929.

[P107] Suriya Subramanian, Michael Hicks et Kathryn S. McKinley. « Dynamic Software Updates : A VM-centric Approach ». In : *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland : ACM, 2009, p. 1–12. ISBN : 978-1-60558-392-1. DOI : 10.1145/1542476.1542478.

[P108] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[P109] The SPaCIFY consortium. *Définition des besoins*. Livrable SPa1. Projet ANR/RNTL SPaCIFY (réf. ANR 06 TLOG 27), fév. 2008.

[P110] Dave Thomas, Damien Boudet et Gérald Garcia. *Cas d'étude initial*. Rapp. tech. Projet ANR/RNTL SPaCIFY (réf. ANR 06 TLOG 27), fév. 2008.

[P111] Ralf Treinen et Stefano Zacchiroli. *Common Upgradeability Description Format (CUDF) 2.0*. Rapp. tech. The Mancoosi project, 2009. URL : http://www.mancoosi.org/reports/tr3.pdf.

[P112] Yves Vandewoude, Peter Ebraert, Yolande Berbers et Theo D'Hondt. « Tranquility : a low disruptive alternative to quiescence for ensuring safe dynamic updates ». In : *IEEE Transactions on Software Engineering* 33.12 (déc. 2007), p. 856–868. DOI : 10.1109/TSE.2007.70733.

[P113] Erwann Wernli, David Gurtner et Oscar Nierstrasz. « Using First-class Contexts to Realize Dynamic Software Updates ». In : *Proceedings of the International Workshop on Smalltalk Technologies*. IWST '11. Edinburgh, United Kingdom : ACM, 2011, 2 :1–2 :11. ISBN : 978-1-4503-1050-5. DOI : 10.1145/2166929.2166931.

[P114] Ji Zhang et Betty H.C. Cheng. « Using temporal logic to specify adaptive program semantics ». In : *Journal of Systems and Software* 79.10 (2006). Architecting Dependable Systems, p. 1361–1369. ISSN : 0164-1212. DOI : 10.1016/j.jss.2006.02.062.

Curriculum Vitæ

État civil

Né le 16 août 1971, 44 ans, Marié, 3 enfants

Contact

Dpt Informatique – Télécom Bretagne
Technopôle Brest-Iroise - CS 83818
29238 Brest Cedex 3, France

Tél : (0|33)2 29 00 14 09
Fax : (0|33)2 29 00 12 82
fabien.dagnat@irisa.fr

Emplois

Maître de conférences en informatique à Télécom Bretagne (2002 -)
ATER au dépt informatique de l'IUT A de Toulouse (2001-2002)
ATER au dépt informatique de l'ENSEEIHT (2000-2001)
Allocataire Moniteur au dépt informatique de l'ENSEEIHT (1997-2000)

Formation

Doctorat en informatique de l'Institut National Polytechnique de Toulouse (2001, mention Très Honorable, prix de thèse de l'INPT)
DEA Informatique Fondamentale et Parallélisme de l'INPT (1997, mention TB, major)
Ingénieur en informatique et mathématiques appliquées de l'ENSEEIHT (1997)

Recherche

Depuis le 1er janvier 2012, ma recherche est réalisée à l'IRISA (UMR 6074) dans le cadre de l'équipe PASS.

J'aborde ou ai abordé les thématiques qui suivent. J'indique également les projets qui ont financé tout ou partie de l'activité, les encadrements liés et le nombre de publications associées.

- La modélisation libre par fédération pour les systèmes critiques complexes (2013-)
  - Projets : ANR Formose, 1 projet CARNOT
  - Encadrements : 1 master, 1 ingénieur
  - Publications : 2 revues, 1 conférence, 1 *workshop*
- Le déploiement de logiciels (2004 -)

  - Projets : CRE FT R&D, 2 projets Institut Télécom, 1 financement de thèse région / chaire cyberdéfense
  - Encadrements : 4 doctorants, 8 masters
  - Publications : 2 revues, 4 conférences, 4 *workshops*
- Les processus de développement pour logiciels adaptables à base de composants (2002-)
  - Projets : 4 projets Institut Télécom, RNTL Accord
  - Encadrements : 3 masters, 1 doctorant, 1 ingénieur, 1 post-doctorant
  - Publications : 3 conférences, 3 *workshops*
- La reconfiguration et les middlewares pour les satellites (2006-)
  - Projets : ANR SPaCIFY, 1 financement de thèse région
  - Encadrements : 4 masters, 1 doctorant, 1 post-doctorant
  - Publications : 1 chap de livre, 2 revues, 4 conférences, 5 *workshops*
- La vérification statique de programmes répartis (1996 - 2002)
  - Thèse sous la direction de P. Sallé et de M. Pantel
  - Encadrements : 1 master
  - Publications : 1 revue, 4 conférences, 2 *workshops*

J'ai effectué un séjour d'études de 3 mois et demi (02/2009 - 05/2009) chez Astrium à Toulouse et un autre de 3 mois et demi (05/2009 - 08/2009) chez Thalès Alénia Space à Cannes. Dans les deux cas, mon rôle a été d'analyser les pratiques de développement logiciel de l'entreprise et de les conseiller dans le cadre d'une évolution vers des approches orientées composants.

Projets

Mes travaux de recherche ont également été possible grâce à divers financements :

- le projet ANR Formose, 2014-2019 ;
- le projet ANR SPaCIFY, 2006-2010 ;
- le Contrat de Recherche Externe France Télécom sauna, 2005-2008 ;
- les projets GET CARISM I en 2003 et II en 2004, CASAC en 2008 et JEMTU 2005-2008 ;
- le projet CARNOT dpan, 2010-2012 ;
- le projet région IMAJD de 2012 à 2015 ;
- le projet du conseil général LINA en 2005.

J'ai participé à la rédaction des dossiers de projet et ai été le responsable Télécom Bretagne pour les projets ANR SPaCIFY et Formose, le CRE FT R&D, les projets instituts, CARNOT et région. J'ai été coordinateur de la pré-proposition ANR Casoar en 2014 qui n'a pas été sélectionnée malgré de bonnes évaluations. Elle sera sans doute re-soumise.

Il convient également d'ajouter la contribution de Télécom Bretagne qui m'a permis de faire un séjour d'étude chez EADS Astrium et Thalès Alenia Space d'un total de sept mois en 2009.

Encadrements

J'ai encadré les doctorants suivants :

- Meriem Belguidoum (10/2004 - 2/2008), maître de conférences à l'université de Constantine en Algérie

- Tuan Anh Trinh (12/2007 - 08/2011), abandon du doctorant pour rentrer au Vietnam, Développeur
- Xu Zhang co-encadré à 50% (09/2010 - 2/2012), démission du doctorant, retour en Chine pour co-gérer l'entreprise familiale de développement logiciel
- Fahad Golra (1/2010-1/2014), post-doctorant à Paris VI

Je dirige ou vais diriger la thèse des doctorants suivants :

- Sebastien Martinez, co-encadré à 75% (10/2012-), la soutenance est prévue pour l'automne
- Bastien Sultan, co-encadré à 50% (10/2015-)

J'ai également encadré les post-doctorants et ingénieurs suivants :

- Jérémy Buisson, (9/2007-2/2009), post-doctorant
- Eveline Kaboré, (9/2008-2/2009), post-doctorante
- Florent Diller, (10/2011-3/2012), ingénieur
- Sylvain Guérin, (2/2015-7/2015), ingénieur

D'autre part, j'ai encadré 17 stages de master recherche ainsi que de nombreux autres stagiaires :

1. David Chemouil, ENSEEIHT (2000), Docteur
2. Kamal Gakhar, Université de Paris XI (2003), Docteur
3. Meriem Belguidoum, Université de Versailles Saint Quentin en Yvelines (2004), Docteur
4. Do Manh Ha, IFI Hanoï (2004)
5. Roméo Said, Télécom Bretagne (2006), Docteur
6. Mohamed Kawtharany, Université de Bretagne Occidentale Brest (2006), Docteur
7. Benoît Vleminckx, Université de Namur (2007), Ingénieur
8. Nguyen Van Hien, IFI Hanoï (2007), Docteur
9. Tuan Anh Trinh, IFI Hanoï (2007), Ingénieur
10. Sebastien Canart, Université de Namur (2008), Ingénieur
11. Cécilia Carro, Université de Buenos Aires (2008)
12. Xu Zhang, Télécom Bretagne (2010), Ingénieur
13. Chafik Merkak, Télécom Bretagne (2011), Ingénieur
14. Chen Xi, Télécom Bretagne (2011)
15. Hanbing Li, Télécom Bretagne (2012), Thèse
16. Mehdi Alaoui Belghiti, Télécom Bretagne (2013)
17. Rachid Ayoubi, École nationale d'ingénieurs de Brest (2014)

Enfin, du fait de mon poste de maître de conférences à Télécom Bretagne, j'ai encadré une douzaine d'étudiants de Télécom Bretagne en stage. Quatre pour des périodes de 6 mois, le reste sur des stages d'été de 2 mois.

Conférences et Expertises

J'ai aidé à l'organisation matérielle de la conférence Euro-Par'99 qui s'est tenue dans les locaux de l'ENSEEIHT. Ainsi qu'à celle d'Euro-TeX 2003 organisée par Yannis Haralambous à Télécom Bretagne. J'ai participé au comité d'organisation des journées Autour du Libre 2004. Enfin, avec Antoine Beugnard, nous avons monté des dossiers de candidature à l'organisation de ECOOP (*European Conference on Object-Oriented Programming*) en 2005 et 2006.

J'ai été relecteur pour la conférence ESOP (*European Symposium on Programming*) en 2009. J'ai été membre du comité de programme et donc relecteur pour HotSWUp 2011 (*Hot Topics in Software Upgrades*) et Lococo (*Logics for Component Configuration*) 2011.

J'ai été expert pour l'ANR de 2011 à 2014 pour des évaluations de proposition mais également pour l'évaluation de projet en cours. J'ai également évalué régulièrement des projets déposés dans le cadre d'appels de l'Institut Mines-Télécom.

Jurys et Comités

J'ai participé au jury de thèse de Guillaume Châtelet le 20 janvier 2006 à l'école Polytechnique.

J'ai participé aux commissions de recrutement de maître de conférences de l'ENSEEIHT en 2009 et l'Université Paris VII en 2011.

J'ai été élu à la commission d'évaluation des appellations de Télécom Bretagne de 2006 à 2012. Cette commission auditionne les candidats aux appellations de maître de conférence et professeur et évalue leur dossier. Elle conseille le directeur sur les promotions vers maître de conférences et professeur et donne son avis sur les recrutements externes.

Avant mon arrivée à Télécom Bretagne, j'ai été membre élu représentant des doctorants et post-doctorants au sein du conseil de laboratoire de l'IRIT (UMR 5505) de 1998 à 2002, membre élu du collège B au CEVU de l'INPT de 1997 à 2001. J'ai également eu un investissement important (secrétaire, webmestre, coordinateur) dans des associations de doctorants locales (AMESAT et CDT) et nationales (CEC). Cela m'a amené à participer au congrès national des doctoriales et à celui des CIES en 1999.

Publications Majeures Récentes

J. Buisson, E. Calvacante, F. Dagnat, S. Martinez et E. Leroux. Coqcots & Pycots : non-stopping components for safe dynamic reconfiguration. In *Proceedings of the International Symposium on Component-Based Software Engineering (CBSE'2014)*. ACM, June 2014.

F. R. Golra, F. Dagnat. Generation of Dynamic Process Models for Multi-metamodel Applications. In *Proceedings of the International Conference on Software and System Process (ICSSP'12)*. IEEE, June 2012.

F. R. Golra, F. Dagnat. The lazy initialization multilayered modeling framework (NIER track). In *Proceedings of the International Conference on Software Engineering (ICSE'11)*. ACM, June 2011.

A. Cortier, L. Besnard, J-P. Bodeveix, J. Buisson, F. Dagnat, M. Filali, G. Garcia, J. Ouy, M. Pantel, A-E. Rugina, M. Strecker, J-P. Talpin. Synoptic : a domain-specific modeling

language for space on-board application software. *Synthesis of embedded software, frameworks and methodologies for correctness by construction.* Springer, 2010, (Engineering), pp. 79-119, ISBN 978-1-4419-6399-4.

J. Buisson, F. Dagnat. ReCaml : execution state as the cornerstone of reconfigurations. *Proceedings of the International Conference on Functional Programming (ICFP'10).* ACM, september 2010, pp. 27-38, ISBN 978-1-60558-794-3.

Responsabilités

J'ai été responsable de la filière *Systèmes Logiciels et Réseaux* de Télécom Bretagne de 2010 à 2015. Il s'agissait d'assurer le suivi des élèves (entre 30 et 40) qui choisissaient l'option informatique en troisième année ainsi que la coordination des différentes UVs. Ainsi, j'ai eu en charge la coordination, les bilans et réflexions pédagogiques. Enfin, j'ai eu à gérer également les stages de fin d'études en entreprise.

En 2002 et 2003, j'ai été responsable de la jeune équipe CADySIMo au sein du département informatique puis co-responsable avec Antoine Beugnard de l'équipe CAMA de 2004 à 2008.

Enseignements

Durant mes cinq ans à Toulouse, puis mes treize ans à Brest, j'ai assuré un grand nombre d'heures d'enseignement et ce dans tous les niveaux (ENSEEIHT, IUT, Télécom Bretagne, Master recherche en informatique) et en formation continue. Durant ces dix-huit années, j'ai assuré, en moyenne, de l'ordre de la centaine d'heures équivalent TD par an. Les principaux sujets de ces interventions sont listés ci-dessous. Les enseignements en gras sont ceux dont je suis ou ai été responsable et contributeur majeur en terme de conception mais aussi de réalisation.

- Algorithmique, Programmation (C, Perl), **Programmation Objet** (Java, Eiffel, C++ et Squeak), **Programmation fonctionnelle** (Caml, Scala et Erlang)
- Génie Logiciel : **méthodes de conception**, **UML**, gestion de version, gestion de projet, test
- **Méthodes formelles** (FSP, λ-calcul, π-calcul)
- Développement avancée : Programmation Web, **Langages**, IHM, **Traduction des langages**, **Systèmes répartis**, **Composant logiciel**, Aspect

Dans le cadre de mon poste à Télécom Bretagne, je participe de manière importante à l'enseignement par projet en encadrant de nombreux projets d'élèves (en moyenne, 1 projet de 1re année, 1 projet de 2e année et 3 projets de 3e année par an).

Depuis 2003, je participe activement à l'enseignement dans le master recherche en informatique. Avec mon collègue Antoine Beugnard, nous avons géré le cours de tronc commun, en visioconférence depuis Brest, MMPFPSD (2004-2007) puis M&F (2012-2015). J'ai aussi participé aux modules d'option SVT (2008-2011) et SIA (2012-2015).

Dans le cadre de la formation continue, j'ai été responsable de deux réponses à des appels d'offre d'Alcatel Business Service sur des formations à la carte. La première formation organisée sur l'année 2007 avait pour objectif d'aligner les connaissances de tous les ingénieurs d'ABS. Il s'agissait de 12 demi-journées de formation sur les sujets suivants : le génie logiciel et l'ingénierie des besoins, la gestion de projet, les méthodes, l'architecture logicielle, la conception

et la programmation objet, les IHM, la qualité, le test, la concurrence et la distribution, la notion de composants, J2EE et les EJB et enfin l'ingénierie dirigée par les modèles. Il y a eu 14 sessions assurées par Antoine Beugnard et moi sur les sites de Brest, Paris et Illkirch. Au total, de l'ordre de 300 ingénieurs ont assisté à cette formation. La seconde formation portait sur le changement de métier de 12 personnes qui basculaient de fonction de conception et réalisation matérielles vers des fonctions de développement. Un cursus sur mesure a été mis en place en 2008, en partie, à Illkirch et en partie à Télécom Bretagne. La moitié des stagiaires a effectué la formation jusqu'au bout et obtenu un certificat. Dans les deux cas, il a fallu gérer les réponses, élaborer les formations, faire les cours, TP et projets et suivre les stagiaires.

J'ai, enfin, été correcteur de l'épreuve d'informatique du Concours Commun Polytechnique de 1998 à 2002.