HAL
open science

Une infrastructure pour l'optimisation de systèmes embarqués évolutifs à base de composants logiciels

Juan Navas

To cite this version:

Juan Navas. Une infrastructure pour l'optimisation de systèmes embarqués évolutifs à base de composants logiciels. Autre [cs.OH]. Université de Bretagne occidentale - Brest, 2011. Français. NNT : . tel-00624826

HAL Id: tel-00624826

https://theses.hal.science/tel-00624826v1

Submitted on 19 Sep 2011

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

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

UNIVERSITÉ DE BRETAGNE OCCIDENTALE
ÉCOLE DOCTORALE SICMA
SANTÉ, INFORMATION-COMMUNICATIONS,
MATHÉMATIQUES, MATIÈRE

T H È S E

pour obtenir le titre de

Docteur

de l'Université de Bretagne Occidentale

Mention Sciences et Technologies de l'Information et de la Communication

Specialité Informatique et Applications

Présentée et soutenue par

Juan F. Navas

Une infrastructure pour l'optimisation de systèmes embarqués évolutifs à base de composants logiciels

Thèse dirigée par Jean-Philippe Babau

préparée au Laboratoire d'Informatique des Systèmes Complexes (LISyC)

soutenue le 6 mai 2011

Jury :

Rapporteurs :	Gilles Muller	-	Directeur de Recherche - INRIA (Regal)
	Laurent Pautet	-	Professeur à Télécom Paristech
Examinateurs :	Antoine Beugnard	-	Professeur à Télécom Bretagne
	Sébastien Gérard	-	Chercheur - CEA (List)
	Stéphane Frenot	-	Maître de conférences (HDR) à l'INSA Lyon (CITI)
	Stéphane Rubini	-	Maître de conferences à l'UBO (LISyC)
	Jacques Pulou	-	Chercheur - Orange Labs
Directeur :	Jean-Philippe Babau	-	Professeur à l'UBO (LISyC)
Invité :	Olivier Lobry	-	Ingénieur de Recherche - CNRS (OREME)

La grandeur des hommes se mesure non seulement par leurs actions, mais aussi et surtout par les actions qu'ils inspirent en leur absence. Sans l'exemple et les leçons qu'il m'a laissé en vie, cette thèse n'aurait jamais pu être achevée. Ce manuscrit est donc entièrement dédié à mon père,
Reinaldo Navas (1950 - 2006).

La grandeza de los hombres se mide no solo por sus actos, sino también y sobre todo por las acciones que estos inspiran cuando ya no están entre nosotros. Sin el ejemplo y las lecciones que me dejó en vida, esta tesis nunca habría podido ser finalizada. Así pues, este manuscrito esta enteramente dedicado a mi padre,
Reinaldo Navas (1950 - 2006).

Résumé :

Concernant la partie logicielle des systèmes embarqués ouverts au changement, on constate (i) le besoin d'un modèle de la structuration du logiciel qui facilite le processus de développement, (ii) la capacité de faire évoluer le comportement du système lors de son exécution, afin de s'adapter aux changements de son environnement, et (iii) la prise en compte des limitations des ressources physiques des plates-formes d'exécution.

L'utilisation de composants logiciels est une approche de conception qui, via l'abstraction des détails techniques, facilite la gestion de la complexité du logiciel. Elle répond à la première des exigences et est à la base de nos propositions. En s'appuyant sur cette approche, la question que nous nous posons est : comment les composants doivent être développés de façon à satisfaire les exigences qui se dérivent des fortes contraintes physiques et du besoin d'évolution du logiciel ?

Pour y répondre, nous introduisons d'abord la notion de *réification* de composant. Une *réification* est une collection de données et de comportements qui encapsulent des aspects du composant à un moment précis de son cycle de vie. Basés sur ce concept, nous proposons ensuite des techniques d'optimisation pour la génération du code «glue» qui assure l'interopérabilité des composants, afin de satisfaire les contraintes physiques dues aux ressources limitées. Ces techniques rendent possible la définition de niveaux d'optimisation pour chaque instance des entités du modèle à composants, ce qui nous permet de régler la performance du logiciel en fonction de ses besoins (extra)fonctionnels. Ensuite, nous définissons quatre catégories d'évolution en fonction des caractéristiques d'évolution de chaque entité du modèle, et nous mettons en place des infrastructures d'évolution adaptées aux besoins d'évolution du système, couplées avec les exigences de performance du logiciel.

Dans la mise en œuvre de nos travaux, nous nous sommes appuyés sur la plate-forme de développement basée composants Fractal/Think. L'efficacité des optimisations proposées et la pertinence des infrastructures d'évolution optimisées ont été validées par des évaluations quantitatives sur la performance du logiciel.

Mots clés : Logiciel embarqué, Évolution, Optimisation, Composants logiciels, Réifications

Abstract :

Regarding embedded systems software which is open to change, we find (i) the need for a model of the structuring of software that facilitates the development process, (ii) the ability to make evolve the system behavior at execution-time, to adapt to changes in its environment, and (iii) the accounting of the physical resources limitations in execution platforms.

The use of software components is a design approach which, through the abstraction of technical details, eases software complexity management. It meets the first requirement and is the basis of our proposals. Based on this approach, we intend to answer the following question : how components should be developed to satisfy the requirements that derive from strong physical constraints and the need of software evolution ?

To answer this question, we first introduce the notion of component *reification*. A *reification* is a collection of data and behaviors that encapsulate aspects of the component at a specific point of its life-cycle. Based in this concept, we then propose optimization techniques for the generation of the glue code that ensures the interoperability of components, in order to satisfy the physical constraints due to limited resources. These techniques make possible the definition of optimization levels for each instance of the component model entities, which allows us to adjust the performance of the software to its (extra)functional needs. Then, we define four categories of evolution based on evolutionary characteristics of each entity of the model, and we deploy an evolution infrastructure tailored to the system's evolution needs, coupled with the software's performance requirements.

In the implementation of our work, we relied on the Fractal/Think component-based development platform. The effectiveness of proposed enhancements and the adequacy of optimized evolution infrastructures have been validated by quantitative assessments on the performance of the software.

Keywords : Embedded software, Evolution, Optimization, Software Components, Reifications

Remerciements

Je suis convaincu qu'un manuscrit de thèse est bien plus que l'aboutissement de trois ans de réflexion autour d'un sujet de recherche, qui donne le mérite à un jeune doctorant d'appartenir à la communauté scientifique. C'est aussi la trace que laisse tous ceux qui ont participé à l'élaboration des pensées ici reflétées, et la volonté des institutions qui l'ont rendu possible. Cette thèse n'est pas l'exception, et j'espère qu'en les remerciant je saurai honorer ces efforts.

Je tiens tout d'abord à remercier mon directeur de thèse, Jean-Philippe Babau, qui s'est toujours montré disponible pour des discussions animées, très à l'écoute et respectueux de mes idées quelques fois obstinées. Son support a été précieux à des moments où cette thèse manquait de moyens pour être achevée, et sans le temps qu'il a consacré à sa relecture, ce manuscrit n'aurait jamais vu le jour.

Mes remerciements s'adressent également à Olivier Lobry, qui a été à l'origine de ce travail de recherche et auprès de qui j'ai énormément appris, d'abord comme stagiaire, puis comme doctorant aux Orange Labs. Je remercie également Jacques Pulou qui, malgré la distance, a su encadrer ce travail et lui donner un caractère industriel. Je remercie dans son ensemble France Telecom pour avoir parrainé ce travail de recherche, au laboratoire NDS de m'avoir accueilli, et en particulier Régine Angoujard et Prosper Chemouil qui m'ont aidé tout au long de la thèse, notamment lors des réorganisations des équipes de R&D aux Orange Labs.

Je tiens à remercier les rapporteurs de ma thèse et les membres du jury pour les échanges enrichissants que nous avons eu lors de la soutenance. J'exprime ma gratitude à tous ceux avec qui j'ai pu confronter mes réflexions : les doctorants, maîtres de conférence et professeurs au laboratoire LISyC à Brest (qui m'ont aussi rendu un service précieux le jour de ma soutenance), les membres de l'équipe SHINE aux Orange Labs Grenoble (Stéphane, Maxime, Matthieu, Julien et Marc, grâce à qui j'ai aussi connu mes limites en tant que skieur...), Frédéric Loiret et les autres participants au projet Flex-eWare, et Tanguy LeFloc'h-Souet qui a été un stagiaire exemplaire.

Enfin, j'adresse mes plus sincères remerciements à ma famille et à mes proches, qui m'ont toujours soutenu, encouragé et supporté (dans tous les sens du mot) tout au long de cette thèse, notamment lors de la rédaction de ce manuscrit. Merci.

Table des matières

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Objectifs de la thèse . . . 3
1.3 Contributions . . . 3
1.4 Organisation du manuscrit . . . 4

2 Composants logiciels pour l'embarqué **7**
2.1 Systèmes embarqués . . . 7
2.1.1 Définition et caractéristiques . . . 7
2.1.2 Éléments de construction . . . 9
2.1.3 Méthodologies de conception . . . 13
2.2 Composants logiciels . . . 14
2.2.1 Introduction . . . 14
2.2.2 Définition . . . 15
2.2.3 Cycle de vie . . . 15
2.2.4 Les composants tout au long du cycle de vie . . . 19
2.3 Synthèse et cadre de la thèse par rapport au domaine . . . 32

3 Optimisation **35**
3.1 Définition . . . 35
3.2 Critères de performance . . . 36
3.2.1 Occupation mémoire . . . 36
3.2.2 Temps d'exécution . . . 36
3.2.3 Consommation d'énergie . . . 37
3.3 Compromis . . . 37
3.4 La performance des composants logiciels et leur optimisation . . . 38
3.5 Optimisation des performances en fonction du cycle de vie . . . 39
3.5.1 Conception . . . 40
3.5.2 Mise en œuvre . . . 40
3.5.3 Exécution . . . 41
3.6 Optimisation dans les plates-formes de développement de référence . 42
3.6.1 Préambule sur l'impact de la compilation de bas niveau . . . 42
3.6.2 TinyOS . . . 43
3.6.3 CAmkES . . . 44
3.6.4 Koala . . . 45
3.6.5 Fractal/Think . . . 45
3.7 Synthèse sur l'optimisation . . . 46

4 Evolution **49**
4.1 Définition 49
4.2 Activités propres à l'évolution 50
4.3 Infrastructures d'évolution 51
4.4 Évaluation des approches d'évolution existantes 52
4.4.1 Considération de l'évolution au niveau des composants 53
4.4.2 Considération de l'évolution en dehors des composants 57
4.5 Synthèse sur l'évolution 61
4.6 Synthèse de la première partie de la thèse 64

5 Réifications de Composants **69**
5.1 Introduction 69
5.2 Réification 70
5.2.1 Définition 70
5.2.2 Propriétés 73
5.2.3 Communication entre réifications 74
5.2.4 Composants miroir 75
5.3 Aperçu de la contribution 76
5.3.1 Réifications pour l'optimisation 77
5.3.2 Réifications pour l'évolution 78

6 Optimisations proposées dans Fractal/Think **81**
6.1 Présentation de la démarche 82
6.2 Attributs 84
6.2.1 Réifications par défaut 84
6.2.2 Coût associé aux réifications 86
6.2.3 Optimisations proposées 87
6.3 Interfaces serveur 89
6.3.1 Réifications par défaut 89
6.3.2 Coût associé aux réifications 91
6.3.3 Optimisations proposées 93
6.4 Interfaces clientes 95
6.4.1 Réifications par défaut 95
6.4.2 Coût associé aux réifications 95
6.4.3 Optimisations proposées 97
6.5 Liaisons 97
6.5.1 Réifications par défaut 97
6.5.2 Coût associé aux réifications 99
6.5.3 Optimisations proposées 101
6.6 Composites 103
6.6.1 Réifications par défaut 103
6.6.2 Coût associé aux réifications 105
6.6.3 Optimisations proposées 107
6.7 Partage de code entre instances de composants 108
6.7.1 Réifications par défaut 108
6.7.2 Coût associé aux réifications 111

6.7.3 Optimisations proposées 111
6.8 Synthèse 112

7 Infrastructures d'Évolution 117
7.1 Évolution dans Fractal/Think 117
7.2 Caractérisation de l'évolution des entités du modèle 120
7.2.1 Évolution dynamique 120
7.2.2 Pas d'évolution 121
7.2.3 Évolution statique 122
7.2.4 Évolution non prédéfinie 125
7.3 Mise en œuvre de l'Évolution non prédéfinie 125
7.3.1 Composants miroir 125
7.3.2 Lien interne entre miroirs et réifications embarquées 133
7.3.3 Réifications embarquées 134
7.4 Infrastructures d'évolution au niveau système 136
7.5 Synthèse 137

8 Evaluation 141
8.1 Plate-forme d'évaluation 141
8.1.1 Motivation et choix 141
8.1.2 Architecture matérielle 143
8.1.3 Logiciel embarqué 145
8.2 Optimisation des composants logiciel 147
8.2.1 Propriétés d'optimisation à la conception 147
8.2.2 Comparaison avec le code *legacy* sur le *bare-metal* 147
8.2.3 Cas d'étude : développement de pilotes de périphériques . . . 151
8.2.4 Cas d'étude : un RTOS à base de composants logiciel 155
8.2.5 Conclusion sur l'évaluation des optimisations 158
8.3 Infrastructures d'évolution 158
8.3.1 Etude de l'impact des Infrastructures d'évolution 159
8.3.2 Infrastructures d'évolution non prédéfinies 161
8.3.3 Conclusion sur l'évaluation des IdE 173

9 Conclusion et Perspectives 175
9.1 Bilan des contributions 175
9.2 Perspectives 177

A Publications liées à la thèse 179

Bibliographie 181

Table des figures

2.1 Couches logicielles classiques des systèmes embarqués 11
2.2 Cycle de vie du déploiement logiciel selon Carzaniga et OSGi. 16
2.3 Cycle de vie simplifié des composants 17
2.4 Une configuration Koala contenant trois composants et un switch côntrolé via une interface de diversité 28

3.1 Vue simplifiée du processus de compilation des plates-formes de développement de référence. 42

4.1 Activités d'évolution et ordre d'exécution 51

5.1 Vue graphique du composant `sound_driver` 71
5.2 Représentation du composant `sound_driver` à la conception 71
5.3 Représentation du composant `sound_driver` après une premiere passe de compilation Fractal/Think 72
5.4 Représentation du composant `sound_driver` à l'exécution 73
5.5 Illustration des liens verticaux, horizontaux et internes entre réifications des composants. 74
5.6 Le composant `sound_driver` et ses réifications tout au long des phases de son cycle de vie. 77

6.1 Vue des entités du modèle Fractal/Think. 82
6.2 Relations existantes entre le contenu des composants et les interfaces du modèle à composants Fractal/Think. 83
6.3 Extraits des réifications du composant `sound_driver` à la conception et après la première phase de compilation. 85
6.4 Extraits des réifications du composant `sound_driver` à la conception et après la première phase de compilation. 90
6.5 Extraits des réifications du composant `test` à la conception et après la première phase de compilation. 96
6.6 Extraits des réifications des composants primitifs `sound_driver` et `test`, à la conception et après la première phase de compilation. . . . 98
6.7 Extraits des réifications du composite `sound_composite` à la conception (haut) et après la première phase de compilation (bas). 104
6.8 Réification de deux instances du même type de composant. 109

7.1 Classe BinAddress. 128
7.2 Classe BinSymbol. 129
7.3 Liens verticaux entre réifications d'un composant. 130
7.4 Classe Entity. 131
7.5 Récupération de l'objet Entity correspondant à l'attribut `maxFreq` du composant `sound_driver`. 133
7.6 Materialisation des liens internes. 135

8.1 Plate-forme Lego Mindstorms NXT 143
8.2 Architecture matérielle de la brique NXT 144
8.3 Architecture simplifiée du logiciel 149
8.4 Structure à base de couches du pilote du coprocesseur 152
8.5 Une application representative de l'embarqué temps-réel 156
8.6 Architecture à composants de l'application 157
8.7 Exécution des tâches lors de trois modifications consécutives de valeurs d'un attribut. 166
8.8 Extrait de la vue architecturale de l'étude de cas du chapitre 8.3.2.4. 170

Liste des tableaux

4.1 Synthèse des techniques qui considèrent l'évolution au niveau des composants . . . 62
4.2 Synthèse des techniques qui considèrent l'évolution en dehors du composant . . . 63

6.1 Variables définies et niveaux d'optimisation des attributs . . . 88
6.2 Description niveaux d'optimisation des attributs . . . 88
6.3 Coût de réification d'un attribut entier par rapport au niveaux d'optimisation. . . . 89
6.4 Variables définies et niveaux d'optimisation des interfaces serveur . . . 93
6.5 Description niveaux d'optimisation des interfaces serveur . . . 94
6.6 Coût de réification des interfaces serveur de l'exemple au chapitre 6.3.2, en fonction du niveau d'optimisation appliqué. . . . 94
6.7 Variables définies et niveaux d'optimisation des liaisons . . . 101
6.8 Description niveaux d'optimisation des liaisons . . . 102
6.9 Coût de réification de la liaison de l'exemple au chapitre 6.5.2 en fonction du niveaux d'optimisation appliqué. . . . 102
6.10 Coût de réification du composite de l'exemple au chapitre 6.6.2, en fonction du niveaux d'optimisation appliqué. . . . 107
6.11 Description niveaux d'optimisation des composites . . . 108
6.12 Variables définies et niveaux d'optimisation des composants multi/-mono instance . . . 112
6.13 Description niveaux d'optimisation du partage de code . . . 113
6.14 Classification des niveaux d'optimisation . . . 114
6.15 Correspondance entre les optimisations étudiées au chapitre 3 et les niveaux d'optimisation proposés. . . . 114

7.1 Interfaces offertes par les composants Fractal/Think. . . . 118
7.2 Questions posées lors de la définition des catégories d'évolution à assigner aux entités du modèle. . . . 121
7.3 Niveau d'optimisation assigné aux entités dans la catégorie Évolution Dynamique. . . . 121
7.4 Niveau d'optimisation assigné aux entités dans la catégorie "Pas d'évolution". . . . 123
7.5 Niveau d'optimisation assigné aux entités dans la catégorie Évolution Statique. . . . 124
7.6 Interfaces requises par les composants miroir. . . . 127
7.7 Interface `ComponentEntities`. . . . 130
7.8 Niveau d'optimisation assigné aux entités dans la catégorie Évolution non prédéfinie. . . . 135
7.9 Dimensions du "comment" de l'évolution et catégories d'évolution proposées. . . . 138

8.1 Propriétés d'optimisation proposées 148
8.2 Description des scénarios d'évaluation 150
8.3 Occupation mémoire des scénarios d'évaluation 150
8.4 Temps d'exécution des méthodes de l'interface LCD 150
8.5 Occupation mémoire - cas coprocesseur 153
8.6 Occupation mémoire - cas écran LCD 154
8.7 Occupation mémoire de l'OS et du système complet en sa version non componentifiée, et surcoût des versions componentifiées 157
8.8 Surcoût en terme du temps d'exécution du chemin considéré. 158
8.9 Occupation mémoire de l'OS et du système complet en sa version non componentifiée, et surcoût de la version componentifiée avec des capacités d'évolution dynamique. 159
8.10 Description des scénarios d'évaluation 160
8.11 Occupation mémoire des scénarios d'évaluation et surcoût des scénarios d'évolution par rapport à la référence. 161
8.12 Occupation mémoire des agents d'évolution. 163
8.13 Services offerts par les composants miroir et opérations d'évolution résultantes de leur invocation. 164
8.14 Temps d'exécution des opérations d'évolution : méthodes `start` et `stop`. 165
8.15 Temps d'exécution des opérations d'évolution : méthode `setAttValue`. 166
8.16 Temps d'exécution des opérations d'évolution : méthode `addSubComponent`de l'interface `ContentController`. 167
8.17 Caractéristiques du système – nombre d'entités du modèle à composants et d'abstractions μC/OS-II 169
8.18 Occupation mémoire du système dans les scénarios définis. 172

Chapitre 1

Introduction

Sommaire

1.1 Motivation 2
1.2 Objectifs de la thèse 3
1.3 Contributions 3
1.4 Organisation du manuscrit 4

Les systèmes embarqués sont aujourd'hui présents dans tous les aspects de notre vie. Nous les retrouvons dans une grande variété de produits, des produits grand public (téléphones portables, appareils ménagers, cadres photo numériques et consoles de jeux), aux équipements de transport (voitures, trains) et aux équipements médicaux et industriels. Pour donner une idée de l'omniprésence des systèmes embarqués, selon une étude récente du Ministère de l'industrie française [Potier 2010], il y a environ 100 milliards de microcontrôleurs actuellement en service dans le monde, à comparer avec le nombre de PC en service, qui était de 1 milliard en 2008.

Dans ce contexte du "tout numérique", le logiciel est un des principaux moteurs d'innovation des lignes de produits industriels. Un nombre grandissant des nouvelles fonctionnalités sont implémentées au niveau logiciel, sans changement des infrastructures matérielles sous-jacentes [1]. La valeur du produit est donc de plus en plus largement portée par le logiciel qui y est embarqué. De même, le coût de fabrication du produit est de plus en plus lié au coût de développement du logiciel. Hélas, la productivité du développement logiciel dans le domaine de l'embarqué reste assez faible (entre 0,5 et 5 lignes de code - LoC - par heure pour le domaine de l'embarqué "critique"), ce qui mène à des surcoûts de production qui pourraient être évités.

L'une des causes à cela est la complexité du processus de développement des systèmes embarqués en général, et du logiciel embarqué en particulier. En effet, il est très différent et significativement plus complexe que le développement de logiciel généraliste. Le logiciel embarqué traite des problématiques liées à son opération dans le "monde réel" [Lehman 1985]. Il est soumis à des contraintes physiques telles que l'utilisation de ressources limitées (mémoire, réseau, processeur, énergie) et des contraintes de qualité de service telles que les contraintes temps-réel, en plus des contraintes classiques du logiciel. Lorsque ces contraintes physiques sont très fortes, elles déterminent en grande mesure les propriétés du logiciel embarqué et du processus de développement.

1. Par exemple, la dernière version du système d'exploitation pour l'iPhone 4 offre plus de 100 nouvelles fonctionnalités (selon l'annonce presse d'Apple consulté le 23 Juin 2010). Il est pour autant possible de l'installer et de l'exécuter sans problème sur l'iPhone 3G, mis en vente sur le marché 2 ans auparavant.

1.1 Motivation

L'expansion récente du marché de dispositifs embarqués, évoquée précédemment, fait que le développement du logiciel embarqué est confronté à des nouveaux défis. Alors que la contrainte de livrer des produits économes en ressources, à bas-coût, reste encore présente, un nombre accru d'exigences doit être satisfait par les fournisseurs de logiciel. Nous en détaillons deux parmi celles identifiées :

- La réduction du délai de mise sur le marché (*Time-To-Market*, TTM), devenu un enjeu central des fabricants qui doivent réagir aussi rapidement que possible aux nouveaux besoins des utilisateurs, ainsi qu'à l'apparition des nouveaux matériels ou technologies, et aux nouvelles normes légales ou de standardisation.
- La capacité d'adapter le fonctionnement du système, déjà déployé et en exécution, au changement de son environnement ou de son contexte d'opération. Par exemple, l'achat et l'installation d'une nouvelle fonctionnalité permet aux dispositifs déjà présents d'offrir de nouveaux services et d'améliorer leur usage.

Ces deux exigences impactent le développement du logiciel embarqué au travers de deux aspects clés :

- La maîtrise du système et de ses composants mène à une réduction du temps de développement et, en lignes générales, à une meilleure gestion de la complexité du logiciel. Dans cet objectif, un modèle clair et fidèle de la structure du logiciel embarqué facilite la vérification de l'exactitude des liens entre les modules constituants, la réutilisation de briques applicatives et l'interaction des différentes partenaires au développement.

- La gestion du caractère dynamique du système se traduit en la capacité de modifier son comportement (et donc celui du logiciel embarqué) une fois le dispositif en fonctionnement. Ce besoin est étroitement lié à la gestion de la maintenance et à l'évolution à l'exécution du logiciel embarqué.

Ces deux besoins sont des manifestations d'un besoin plus général : la flexibilité, définie comme la souplesse du logiciel exprimée tout au long de son cycle de vie, soit la capacité de modifier une sous-partie du système tout en conservant une cohérence fonctionnelle globale. En résumé, on demande aux acteurs du marché des logiciels embarqués d'inclure plus de flexibilité dans leur produits.

Dans l'étude de l'état de l'art du domaine, nous avons noté que les efforts conduisant à une flexibilité forte du logiciel embarqué ne considèrent pas les aspects qui différencient le logiciel embarqué des autres logiciels, i.e., la prise en compte de contraintes physiques fortes et notamment des contraintes sur l'utilisation des ressources matérielles limitées. De plus, nous constatons que les travaux qui se focalisent sur la gestion économe des ressources physiques disponibles ne considèrent pas la perte, en terme de flexibilité (spécialement à l'exécution du logiciel), que leurs propositions entraînent.

1.2 Objectifs de la thèse

Dans ce contexte, l'objectif de cette thèse est de réconcilier le besoin de flexibilité des logiciels embarqués avec les contraintes propres à ce genre des systèmes, notamment la limitation des ressources physiques disponibles. Vu la généralité de cet objectif, nous avons centré nos travaux de recherche, autour de la question de la « flexibilité vs. ressources limités » au niveau de l'exécution pour des logiciels construits à base de composants.

- D'abord, nous nous intéressons aux approches de conception du logiciel embarqué qui facilitent la gestion de la complexité du logiciel par l'abstraction des détails techniques, via une structuration "gros grain", puis le raffinement de ces structures. Nous nous focalisons sur les approches basées sur les **composants logiciels** [Szyperski 2002], qui mettent l'accent sur la séparation des préoccupations liées au matériel et au logiciel. Sur ceux-ci, cette thèse vise à proposer des techniques d'**optimisation** afin de combler les lacunes de ces approches à haut-niveau d'abstraction sur le niveau d'utilisation des ressources physiques. Les optimisations devront agir sur l'implémentation de la « glue » qui réifie les assemblages de composants à l'exécution, et non sur le comportement de ceux-ci. L'idée est ici de rester relativement indépendants des caractéristiques des applications, très hétérogènes dans le domaine de l'embarqué.
- Ensuite, nous nous intéressons à l'**évolution à l'exécution** du logiciel embarqué, et notamment à l'implémentation des infrastructures d'évolution (IdE), i.e. les parties du logiciel embarqué qui permettent l'exécution des activités d'évolution : observation, raisonnement et intervention. Nous nous focalisons sur le compromis que l'on constate entre la richesse d'évolution fournie par l'IdE et l'impact de celle-ci sur la performance du logiciel embarqué et donc sur l'utilisation des ressources physiques disponibles. Sur ce point, la thèse vise à proposer des catégories d'évolution qui peuvent être assignées à chaque instance des entités du modèle à composants, pour ensuite générer des IdE optimisées et adaptées aux besoins d'évolution de chaque entité.

1.3 Contributions

Pour répondre à ces objectifs, notre travail apporte les contributions suivantes.

- Nous introduisons la notion de **réification** de composant. Une réification d'un composant est une collection de données et de comportements qui encapsulent un ou plusieurs aspects spécifiques du composant à un moment précis de son processus de développement.

  Nous utilisons les réifications comme abstractions « pivot » afin d'assurer une traçabilité des composants tout au long de leur cycle de vie, dès leur conception jusqu'à son exécution. Nous sommes ainsi en mesure d'associer les décisions prises par rapport à l'optimisation et à l'évolution du logiciel, et de maîtriser le compromis entre la richesse d'évolution et la performance du logiciel.

- Nous proposons des techniques d'optimisation du code « glue » des composants et à partir de celles-ci nous définissons des niveaux d'optimisation pour chaque entité du modèle à composants. Ces techniques ont été présentées dans [Lobry 2009], où une brève description de leur mise en œuvre (détaillée dans cette thèse) est fournie. Nous avons voulu appliquer ces techniques sur des cas d'utilisation typiques du domaine de l'embarqué dans des plates-formes matérielles diverses, ce qui a été fait dans [Loiret 2009] par rapport aux systèmes d'exploitation temps réel, et dans [Navas 2009a] en ce qui concerne le développement de pilotes de périphériques.
- Dans [Navas 2009b] nous avons constaté le nécessaire compromis entre richesse d'évolution et performance du logiciel, ainsi que l'inaptitude des IdE existantes pour gérer un bon nombre de cas d'évolution. Nous proposons des catégories d'évolution pour chaque entité du modèle à composants, et mettons en place des IdE adaptées aux besoins d'évolution des entités, et optimisées par rapport à l'utilisation de ressources physiques. Ces propositions ont été publiées dans [Navas 2010a] et [Navas 2010b].

1.4 Organisation du manuscrit

Cette thèse est organisée en trois parties. Après l'introduction, une première partie de la thèse traite du contexte de l'étude et de l'état de l'art. Elle est découpée en trois chapitres :

Le **chapitre 2** décrit le contexte de cette thèse, soit le domaine des systèmes embarqués et des composants logiciels, et définit le cadre des travaux de recherche. Nous présentons également quatre plates-formes de développement logiciel à base de composants pour les systèmes embarqués, qui nous semblent représentatives du domaine, et qui seront systématiquement citées aux chapitres 3 et 4.

Le **chapitre 3** présente un état de l'art sur les techniques d'optimisation du logiciel. Après une définition de ce que le mot "optimisation" signifie dans le cadre de cette thèse et des critères d'optimisation qui nous intéressent, nous évaluons les techniques d'optimisation offertes par les plates-formes de développement logiciel à base de composants représentatives du domaine.

Le **chapitre 4** présente un état de l'art sur l'évolution à l'exécution du logiciel embarqué. Nous définissons l'évolution à l'exécution et les IdE. Nous définissons les critères qui caractérisent la performance des IdE et évaluons les approches d'évolution existantes par rapport à ces critères. Parmi les approches évaluées on retrouve les plates-formes de développement logiciel à base de composants représentatives du domaine.

Dans une deuxième partie, nous présentons nos contributions au travers de trois chapitres :

Le **chapitre 5** définit et caractérise la notion de *réification*, qui est à la base des propositions des chapitres 6 et 7. A la fin de ce chapitre nous présentons un aperçu de la contribution et comment elle utilise la notion de réification.

Le **chapitre 6** présente les techniques d'optimisation proposées. Pour chaque entité du modèle à composants nous définissons des niveaux d'optimisation pour lesquels on applique des techniques d'optimisation plus ou moins agressives. Des détails sur la construction (à la compilation) des réifications à l'exécution sont fournis.

Le **chapitre 7** définit les catégories d'évolution qui sont applicables aux entités du modèle à composants, et propose des infrastructures d'évolution adaptées et optimisées en fonction des caractéristiques d'évolution représentées par ces catégories. La mise en œuvre des liens entre réifications du même composant à différents instants du cycle de vie logiciel est ici détaillée.

Enfin, dans une troisième partie nous évaluons les propositions avant de conclure :

Le **chapitre 8** présente les évaluations menées sur les propositions faites aux chapitres précédents, ainsi que les plates-forme matérielles et logicielles utilisées dans ces évaluations. Il permet de caractériser l'approche en terme quantitatif.

Le **chapitre 9** présente un sommaire du travail mené dans cette thèse. Il signale les directions sur lesquelles d'autres travaux peuvent être effectués dans le futur.

Chapitre 2

Composants logiciels pour l'embarqué

Sommaire

2.1 Systèmes embarqués 7
2.1.1 Définition et caractéristiques 7
2.1.2 Éléments de construction 9
2.1.3 Méthodologies de conception 13
2.2 Composants logiciels 14
2.2.1 Introduction 14
2.2.2 Définition 15
2.2.3 Cycle de vie 15
2.2.4 Les composants tout au long du cycle de vie 19
2.3 Synthèse et cadre de la thèse par rapport au domaine . . . 32

Ce chapitre vise à décrire le contexte de notre étude, soit le domaine des systèmes embarqués et des composants logiciels. Dans un premier temps nous présentons les systèmes embarqués visés dans l'étude dont nous identifions les caractéristiques et les éléments de constitution qui nous intéressent tout particulièrement. Ensuite, nous nous focalisons sur le logiciel embarqué et notamment sur l'approche de développement à base de composants logiciels. Enfin, nous présentons quatre plates-formes de développement à base de composants logiciels pour les systèmes embarqués qui nous semblent représentatives du domaine ; elles serviront d'éléments de référence pour l´état de l´art sur l'optimisation et l'évolution.

2.1 Systèmes embarqués

2.1.1 Définition et caractéristiques

Henzinger et Sifakis définissent un système embarqué comme un artefact d'ingénierie impliquant des calculs qui sont soumis à des contraintes physiques de deux types : une première contrainte concerne la réaction vis-à-vis d'un environnement physique, et une deuxième contrainte concerne son exécution sur une plate-forme physique - électronique [Henzinger 2006]. Cette définition s'adapte convenablement à la grande hétérogénéité des systèmes embarqués que l'on constate de nos jours et, malgré sa généralité, elle nous permet d'identifier deux caractéristiques clé des systèmes embarqués actuels qui sont l'orchestration des processus physiques et informatiques, et l´enfouissement dans un objet concret.

L'orchestration des processus physiques et informatiques. Le logiciel embarqué est étroitement lié à des processus physiques : il surveille et contrôle des processus physiques à l´aide de boucles de rétroaction qui exercent une influence sur les processus physiques via des calculs. Ceci motive la proposition de méthodologies de conception hybrides qui prennent en considération ces deux facettes (physique et logicielle) des systèmes embarqués. La sous-discipline connue comme *Cyber-Physical Systems*, CPS [Lee 2006, Lee 2008], de récente apparition, cherche à explorer les problématiques de recherche liées à cette propriété.

L'enfouissement dans un objet concret. L'existence d'un système embarqué est liée à l'existence d'un objet physique dans lequel il est enfoui, et dont une partie de ses fonctionnalités (sinon toutes) sont surveillées ou contrôlées par celui-ci. Par conséquent, le système embarqué se voit confronté à des contraintes associées à la construction et les usages des tels objets (taille, prix de production, autonomie énergétique).

Les caractéristiques précédemment énoncées ont un effet sur la nécessaire sûreté de fonctionnement des systèmes et sur l´utilisation des ressources matérielles :

- **Sûreté de fonctionnement**. Afin de mesurer l´état du système contrôlé, les systèmes embarqués visent à assurer à tout prix la sûreté de fonctionnement sous des conditions d'opération variables (parfois extrêmes). Cela se traduit par le respect des contraintes fonctionnelles mais aussi temporelles pour suivre l´évolution du processus physique et réagir en temps contraint. Concernant la criticité de fonctionnement, une preuve formelle sur la capacité du logiciel à assurer ce respect est requise. De même, pour les objets embarqués du quotidien, l'impact économique d'un dysfonctionnement impose de respecter des contraintes de sûreté de fonctionnement.
- **Utilisation de ressources**. Les contraintes propres aux objets physiques auxquels les systèmes embarqués sont intégrés, impliquent, à un degré plus ou moins élevé, des contraintes sur l'utilisation des ressources matérielles limitées (la mémoire, les processeurs et les médiums de communication). On exige de plus la prédictibilité sur l´utilisation des ressources afin de gérer la sûreté de fonctionnement et assurer une gestion économe des ressources, prenant en compte leur disponibilité.

En plus de ces contraintes classiques du domaine, actuellement les nouveaux usages donnés aux objets enfouis exigent des systèmes embarqués une ouverture au changement.

- **Ouverture au changement**. Les systèmes embarqués ont été traditionnellement considérés comme des logiciels fermés au monde extérieur, exécutant une ou plusieurs taches bien précises et exposant un comportement avec des plages de variation bien définies. La miniaturisation et l´intégration des systèmes embarqués dans les objets du quotidien, ainsi que l´expansion de la couverture des réseaux de communication filaires et sans fils, ont rendu possible de nouveaux contextes d'opération où l´évolution logicielle post-déploiement est nécessaire.

Dans cette thèse nous nous intéressons à la construction de systèmes embarqués

ouverts au changement, et dans ce contexte nous nous focalisons sur l´utilisation contrainte de ressources matérielles limitées.

Nous présentons maintenant les éléments de construction des systèmes embarqués, en insistant sur les aspects logiciels.

2.1.2 Éléments de construction

Les systèmes embarqués sont constitués de logiciel et d'une plate-forme matérielle sous-jacente. Dans le cadre de cette thèse nous nous intéressons au logiciel embarqué contraint par des ressources matérielles limitées.

2.1.2.1 Matériel

Comme tout système informatique, un système embarqué comprend, au minimum (et souvent au maximum) une unité de calcul et de la mémoire. A ces deux éléments primordiaux peuvent être ajoutés, pour les besoins de communication et de traitement spécifiques, toute sorte de composants matériels (décodeurs audio/vidéo, convertisseurs analogue/numérique, piles de communication...) qui interagissent avec l'unité de calcul à travers des ports d'entrée/sortie.

- **Unités de calcul**. Elles sont choisies en fonction de trois critères principaux : le coût par unité, la consommation d'énergie et la puissance de calcul. Leurs performances vis-à-vis de ces critères sont limitées par la fréquence maximale d´horloge, le nombre de cycles par instruction (CPI), le nombre d´instructions disponibles et la taille de chaque instruction : 4, 16, 32 ou 64 bits. En conséquence, ces caractéristiques fixent des limites aux performances obtenues par le logiciel qui s´exécute sur ces unités de calcul. On distingue, en fonction du jeu d´instructions utilisé, des familles d´unités de calcul : 80x86 [Agarwal 1991], PIC [Mazidi 2009], ARM [Seal 2000], AVR [Barrett 2007], MIPS [Kane 1988], Motorola 6800 [Motorola 1989], PowerPC [Bunch 1996], entre autres.
- **Mémoire**. La mémoire est tout composant électronique permettant de stocker temporairement des données. Ils se différencient principalement par rapport à leurs capacités de stockage, aux caractéristiques d'accès aux données (débit, temps d'accès et temps de cycle), à la volatilité des données et au prix par unité de stockage. Les types de mémoire les plus représentatifs sont :
    - **Random Acces Memory (RAM)**. Elle permet la lecture et l´écriture de données. Elle est constituée d´un grand nombre de condensateurs qui représentent des bits de la mémoire et dont la charge électrique détermine l´état logique et donc les informations stockées. Lorsque la mémoire est hors tension, les données stockées sont perdues ; pour cette raison elle est appelée mémoire volatile. Par ces caractéristiques, ainsi que par son temps d´accès réduit, elle est utilisée pour la lecture et l´écriture de données fréquemment manipulées et non persistantes, et en particulier pour la sauvegarde de l´état courant du programme en exécution. Son prix relativement élevé limite néanmoins son utilisation.

- **Read Only Memory (ROM)**. Elle permet l'écriture de données une seule fois, et la lecture de données autant de fois que nécessaire. Contrairement à la RAM, ce type de mémoire conserve des données en l´absence de courant électrique et elle est appelée mémoire non volatile. Ces caractéristiques la rendent particulièrement adaptée pour le stockage d´informations persistantes et non modifiables, comme celles nécessaires au démarrage des systèmes embarqués tels que le système basique d´entrée/sortie (BIOS) et les instructions non modifiables comme les politiques de sécurité. Puisque le temps d'accès d'une ROM est beaucoup plus lent que celui d'une RAM (150ns vs. 10ns pour une SDRAM en moyenne [1]), les instructions contenues dans la ROM sont souvent copiées en RAM au démarrage, par une procédure appelée *shadowing*.
- **FLASH**. Cette mémoire est un compromis entre les deux précédentes : elle est non-volatile comme la ROM et elle permet l'écriture fréquente de données comme la RAM (quoique plus lente). Par conséquent, elle est utilisée pour stocker des informations persistantes et modifiables. L´écriture en FLASH est limitée en nombre et en granularité : l´unité d´écriture est une page (256 bytes) et chaque page peut être réécrite de 10,000 à 100,000 fois en fonction de la technologie utilisée. Ce type de mémoire est utilisé pour stocker les exécutables (application et exécutif) et les données qui n´ont pas vocation à être modifiées très fréquemment ou à des granularités très fines, tels que les données de configuration et les fichiers.

- **Ports d'entrée/sortie**. Ils permettent d'interagir avec des périphériques externes : interfaces homme-machine (IHM), réseaux, capteurs et actionneurs. Parmi les plus populaires on trouve des convertisseurs analogue-numérique (ADC), les multiples variations des ports série : *Serial Peripheral Interface* (SPI), *Two-wire Interface* (TWI), Universal Serial Bus (USB), *Inter Integrated Circuit* (I2C), ainsi que les liaisons réseau *Ethernet* et *Zigbee*. Pour chaque port d´entrée/sortie et, dans un plus haut niveau d´abstraction, pour chaque périphérique lié à l´unité de calcul, un logiciel appelé *pilote* assure la gestion et la communication avec le périphérique. Ce logiciel est étroitement lié aux caractéristiques matérielles du port d´entrée/sortie.

Dans cette étude nous visons des machines du type *microcontrôleurs*. Celles-ci intègrent dans un seul boîtier et à un coût réduit les trois composants matériels décrits précédemment avec des configurations diverses, ainsi que d'autres éléments non cités précédemment comme les compteurs et les canaux PWM (*Pulse Width Modulation*).

D´autres solutions adaptées à des besoins très concrets, tels que les circuits intégrés spécifiques aux applications (ASIC), les processeurs dédiés au traitement de signaux numériques (DSP) ou les circuits logiques programmables (FPGA) ne sont pas considérées dans cette étude.

1. Le temps d'accès dépend fortement du temps de cycle, déterminé par la fréquence d'horloge de l'unité de calcul, et du temps de latence propre à la mémoire. Sa valeur précise ne peut donc être déterminée qu'en fonction de ces deux facteurs.

2.1.2.2 Logiciel

Par rapport à d'autres systèmes informatiques, où une séparation très marquée entre le logiciel et l'environnement physique a permit la production massive et rapide de code, le développement de systèmes embarqués et plus complexe. Afin d'intégrer les contraintes matérielles et la sûreté de fonctionnement, les développeurs de logiciels embarqués doivent prendre en considération des aspects liés aux caractéristiques des plates-formes matérielles d'exécution.

Par des raisons de facilité de programmation et de réutilisation, ces considérations sont classiquement encapsulées par un *exécutif* et les pilotes associés aux périphériques, alors qu´un *applicatif* regroupe le logiciel qui est en plus grande mesure indépendant des plates-formes matérielles, comme le montre la figure 2.1. Ces deux couches logicielles interagissent afin d´assurer les calculs liés aux contrôle des processus physiques. En conséquence, le développement d´un système embarqué se doit de considérer les deux couches logicielle, l´exécutif et l´applicatif.

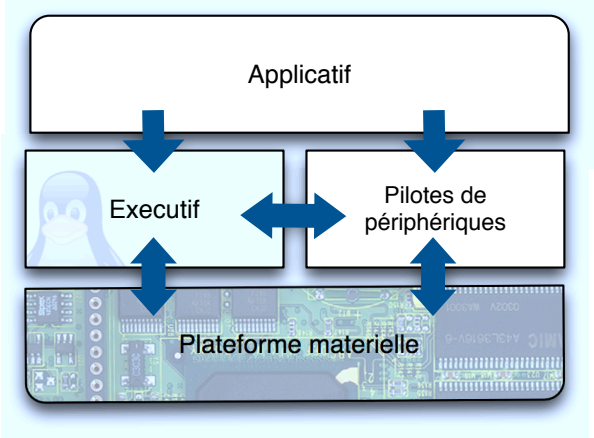


FIGURE 2.1 – Couches logicielles classiques des systèmes embarqués

Exécutif et Pilotes de périphériques. Ce niveau assure la gestion des composants matériels de la plate-forme au travers des pilotes des périphériques. Il est chargé également de la gestion des diverses applications qui sont exécutées par le processeur et de la communication entre celles-ci. Il peut aussi inclure des services communs tels que les systèmes de fichiers ou les protocoles réseau. Les considérations liées au respect de limites ou des contraintes temporelles (aspects temps-réel) sont aussi traitées à ce niveau.

Applicatif. Ce niveau, comme son nom l'indique, concerne le logiciel qui correspond aux applications et qui implémente les fonctionnalités de contrôle des processus physiques et de communication demandées au système embarqué. Le plus souvent, les applications correspondent à des processus, des tâches ou des *threads* (suivant les conventions choisies dans l´exécutif) qui sont traités de fa-

çon concurrente par le processeur, la concurrence étant gérée au niveau de l´exécutif.

La frontière entre ces deux couches, ainsi que la structuration interne de chacune d´elles, est une source des nombreuses polémiques et le sujet d´un grand nombre de travaux de recherche. En particulier, au niveau de l´exécutif des diverses familles on fait surface : noyaux monolithiques modulaires [Ford 1997] et non modulaires [Ritchie 1986], micro-noyaux [Rozier 1991, Krieger 2006, Bomberger 1992], exo-noyaux [Engler 1995]. Dans le cadre de cette étude ce débat a une pertinence limitée, nous orientons donc le lecteur vers les références dans le cas où le sujet attire son attention.

Nous nous intéressons maintenant à l´exécution du logiciel embarqué.

Exécution

Nous distinguons deux stratégies pour l´exécution du logiciel embarqué : la projection du code sur une infrastructure logicielle permettant son exécution, et la compilation du code vers un langage intermédiaire exécuté par la plate-forme matérielle sous-jacente.

Dans la première stratégie l´installation, l´activation et l´exécution du logiciel sont gérés par un intergiciel dédié. C'est le cas des machines virtuelles comme PERC [AonixPERC 2010], Lejos [LejOS 2010b] et JavaCard [Microsystems 2010], et des plates-formes de services comme OSGi [The OSGi Alliance 2007]. Les machines virtuelles apportent des solutions intéressantes par rapport à l´ouverture au changement comme le téléchargement dynamique de code. La quantité de ressources physiques nécessaires pour déployer ces infrastructures d´exécution limite leur utilisation dans le domaine des systèmes embarqués. Les solutions existantes à ce jour ne sont pas portables sur une cible du type microcontrôleur car trop gourmandes en ressources (PERC), ou n´offrent pas les capacités d´évolution logiciel visées dans cette thèse (Lejos), ou encore sont très spécifiques à un domaine et n'offrent pas le téléchargement dynamique de code (Lejos, JavaCard). Pour autant, nous reviendrons sur les solutions à base de machines virtuelles en fin de thèse.

La deuxième stratégie correspond à la compilation classique du code source en langage assembleur, qui est ensuite traité par l´unité de calcul. Dans le cadre de cette thèse, nous nous focalisons sur les systèmes appartenant à cette dernière catégorie.

Langages

A l'heure actuelle, 60% des développeurs dans le domaine des systèmes embarqués utilisent les langage C [Kernighan 1988, Institute 2002] et 20% le langage C++ [Stroustrup 2000], selon une étude récente [EE Times Group 2010]. Si l'on considère qu'une bonne partie de ceux qui utilisent C++ ne tire pas profit des caractéristiques « orienté objet » du langage, on trouve que près de 80% des logiciels embarqués sont basés sur la famille des langages C.

Le bas niveau d'abstraction sur les architectures matérielles que le langage C offre, permet aux développeurs de s'adapter de façon efficace aux caractéristiques

des plates-formes physiques. Dans le cadre de cette thèse nous nous intéressons aux applications développées en langage C.

En conclusion, nous nous intéressons aux systèmes embarqués dont le code est écrit en langage C, compilé en langage assembleur et ensuite exécuté sur un boitier de type microcontrôleur, intégrant une unité de calcul, des mémoires physiques de type RAM, FLASH et/ou ROM, et des périphériques variés. Nous supposons que les infrastructures d´exécution sont inexistantes ou minimales, i.e. qu´elles nous permettent de traiter des aspects logiciels très proches du matériel. Par conséquent, nos travaux s´appliquent aussi bien aux couches basses (exécutifs et pilotes de périphériques) qu´aux applications. Les méthodologies de conception logiciel de tels systèmes font l´objet du chapitre suivant.

2.1.3 Méthodologies de conception

Le logiciel embarqué a été traditionnellement conçu en suivant des méthodologies *centrées sur le code source*, qui mettent l´accent sur l´activité de programmation ou "codage". Cette approche permet de maîtriser la façon dont le logiciel utilise les ressources physiques, notamment la mémoire ; elle permet aussi de maîtriser en détail le comportement temporel du logiciel et assurer la prédictibilité sur l´utilisation des ressources.

Néanmoins, elle n´est pas facilement extensible à la conception de systèmes logiciels, i.e. des logiciels composés des nombreux modules conçus par des acteurs différents [DeRemer 1975]. Cette approche n´est pas adaptée aux contraintes plus récentes de la construction des systèmes embarqués : la complexité croissante des logiciels, le nombre en hausse des acteurs qui interviennent dans le processus de développement et le degré d'interaction entre ceux-ci, la réduction du temps de mise sur le marché et par conséquent le besoin de réutilisation massive de code source.

Afin de faire face à ces enjeux, un certain nombre d'approches de développement à plus haut niveau d'abstraction ont été proposés ces dernières années. Parmi celles-ci, on retrouve :

L'ingénierie basée sur les composants logiciels (*Component-Based Software Engineering*, CBSE) [Szyperski 2002] permet la conception de systèmes via l'assemblage structurel de composants, soient des entités de conception indépendantes. Elle met l'accent sur la séparation des préoccupations, comme celles liées au matériel et au logiciel. Les architectes décrivent ces assemblages de composants avec des langages dédiés de description d'architectures (*Architecture Description Languages*, ADL).

L'ingénierie dirigée par les modèles (IDM) a pour objectif de permettre le développement de logiciels en utilisant des modèles, soit des représentations simplifiées d'un ou plusieurs aspects du système. En particulier, des modèles indépendantes aux plate-formes d'exécution (*Platform Model*, PM) et des modèles dépendantes à celles-ci (*Platform Independent Model*, PIM) peuvent être proposés, à partir desquels un modèle spécifique à la plate-forme et à l'application est obtenu [OMG 2003, DeAntoni 2005]. La transformation de modèles est une activité centrale

des approches IDM.

Les approches à plus haut niveau d'abstraction facilitent l'interopérabilité, la réutilisation de code, le déploiement sur des plates-formes matérielles diverses et la gestion de la documentation, et en général tout le processus de développement de systèmes logiciels. Leur efficacité par rapport aux approches plus classiques a été démontrée, notamment dans les domaines des systèmes critiques et des télécommunications [Krasner 2008].

Dans cette étude, nous adoptons cette famille de méthodologies de conception, et plus particulièrement à base de modèles de composants logiciels. Les bénéfices attendus de cette démarche sont :

- La maîtrise de la complexité du logiciel, via l´abstraction des détails techniques via une structuration "gros grain", puis le raffinement de ces structures. Cette approche facilite la communication entre les différents acteurs du développement du système en leur fournissant un niveau de détail adapté à leurs tâches [DeRemer 1975, Bass 1997].
- La facilitation de la validation du système, en appliquant des techniques formelles de vérification sur les diverses parties du système et sur leur composition, à différentes niveaux d´abstraction [Garlan 2003, Bouyssounouse 2005, Gössler 2007, Poulhies 2010].
- Un cadre d´étude et de mise en œuvre de l´évolution logicielle qui facilite la détermination des parties des systèmes à être modifiées, le raisonnement sur les conséquences du changement et le contrôle sur le processus de changement afin de préserver l´integrité du système [Oreizy 1998, Gar 2003, Stuckenholz 2005, Oreizy 2008]. La notion de hiérarchie (cf. 2.2.4.1 page 22) permet de manipuler des blocs de logiciels sans limitation de taille, qui ont à la fois un sens fonctionnel et qui ont un sens du point de vue de l'ingénierie logicielle.

Les composants logiciels pour l´embarqué font l´objet du chapitre suivant.

2.2 Composants logiciels

2.2.1 Introduction

Au fur et à mesure que les disciplines de l'ingénierie sont devenues matures, elles ont commencé à manipuler des éléments de plus en plus sophistiqués. L'idée de construire des nouveaux éléments par l'assemblage d'unités moins complexes et bien connues s'est imposé comme moyen pour faciliter le développement et l'analyse des systèmes. Cela est le cas pour des branches comme l'ingénierie civile, mécanique et chimique, et notamment pour l'électronique, où à titre illustratif l'invention en 1958 du circuit intégré (*Integrated Circuit*, IC) [Kilby 1964] a permit de regrouper des composants électroniques (transistors, condensateur, résistances) et reproduire une ou plusieurs fonctions électroniques plus ou moins complexes et bien définies.

L'ingénierie du logiciel n'a pas été l'exception. Déjà en 1968, M.D. McIlroy [McIlroy 1968] proposait les *composants logiciels* comme briques de construction de routines et d'algorithmes dans la production en masse de logiciels. Certaines des

retombées telles que la réutilisation des composants sur étagère (*components off-the-shelf*, COTS) sont d'ores et déjà envisagées dans ses travaux. L'adoption à large échelle du paradigme des composants logiciels a néanmoins été lente, et ce n'est que dans les années 1990 que les praticiens ont commencé à en entendre parler, notamment avec l'introduction des *Enterprise JavaBeans* (EJB) [Burke 2006], et des modèles à composants COM de Microsoft [Microsoft 1999] et le *CORBA Component Model* (CCM) de l'OMG [OMG 2010].

2.2.2 Définition

Donner une définition précise et unique du terme composant logiciel devient une tâche complexe et pour certains même impossible [Grone 2005], du fait que la définition varie en fonction du contexte d'utilisation. Il existe néanmoins un consensus autour de la définition proposée en 1996 lors du premier atelier de travail consacré à la programmation orientée composants [WCO 1996] :

« A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties »

Cette définition, même si elle demeure très générale, introduit un ensemble de concepts qui seront traités plus en profondeur dans les chapitres suivants. Elle met l'accent sur la composition en tant que paradigme de conception et d'implantation d'applications, et implicitement sur la réutilisation de composants logiciels dans des cas d'utilisation diverses. Le rôle des interfaces comme seul moyen d'exprimer les dépendances des composants envers son environnement est aussi souligné. D'ailleurs, le fait que rien ne soit dit à propos de leur implémentation donne une image des composants comme étant des « boites noires » décrites par des interfaces [2], image qui peut s'avérer trompeuse.

Finalement, en considérant les composants comme unités « destinées à être déployées », cette définition fait une référence (certes timide) à des étapes du cycle de vie logiciel autres que la conception. Dans les systèmes embarqués, l'utilisation des ressources étant au coeur des préoccupations, les composants doivent être étudiés à tous les niveaux du cycle de vie, en particulier lors des étapes "basses" (compilation et déploiement) qui ont un impact fort sur l'utilisation des ressources. Le cycle de vie des composants logiciels est l'objet du chapitre suivant.

2.2.3 Cycle de vie

Le cycle de vie d'un composant [Crnkovic 2006] détermine l'ensemble d'activités de développement nécessaires pour arriver à son exécution, à partir de la description en langage humain de son comportement. Les modèles de cycle de vie proposés, tels

2. Cette image des composants comme des « boites noires » décrites par ces interfaces et prêtes à être déployées dans un contexte d'opération quelconque, même si elle répond à l'idéal vis-à-vis le modèle économique imaginé pour les composants logiciels, ne correspond que rarement à la réalité. Dans la pratique on trouve le plus souvent des composants «boites blanche » ou « boites grises » en fonction du degré d'expositions de son implémentation.

que le cycle en V [Forsberg 1991, iABG 1992], le cycle itératif [Larman 2003] ou le cycle en cascade [Royce 1987], définissent un ensemble d'**étapes** pour le processus de développement du logiciel, ainsi que des **activités** interdépendantes menant le logiciel d'une une étape à une autre. Ces éléments sont souvent représentés via un diagramme état/transitions. La définition et le niveau de détail des étapes, ainsi que la terminologie utilisée, varie en fonction du modèle et de l'utilisation donnée à celui-ci [3].

Certains travaux se focalisent sur une ou plusieurs étapes particulières du cycle de vie, et proposent des modèles détaillés des activités propres à cette étape. Prenons le cas du déploiement logiciel [Dea 2007] : dans [Lestideau 2002] et [Carzaniga 1998], le déploiement des composants est décrit par rapport aux activités qui sont menées. Certaines activités comme la mise à jour et l'adaptation, que l'on pourrait considérer plus liées à l'exécution du logiciel, sont considérées comme propres au déploiement (cf. figure 2.2, gauche). La description du cycle de vie des *bundles* OSGi [The OSGi Alliance 2007] est faite en fonction des états possibles du logiciel, les activités qui mènent d'un état à un autre sont similaires à celles du premier cas. Le cycle de vie dans OSGi introduit la notion de résolution des dépendances des *bundles* vis-à-vis de la plate-forme d'exécution sous-jacente (cf. figure 2.2, droite). Ces deux cas illustrent l'importance des résolution des dépendances et la gestion et adaptation des composants en dehors de sa simple exécution.

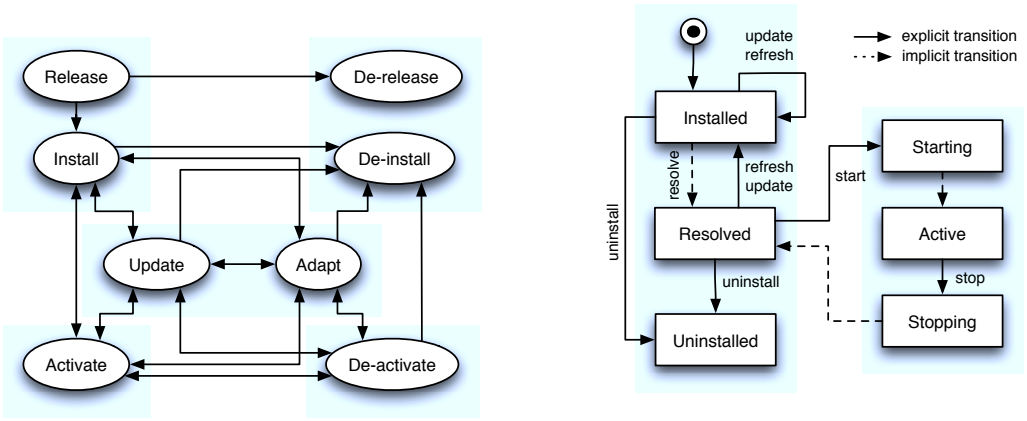


FIGURE 2.2 – Cycle de vie du déploiement logiciel selon Carzaniga (gauche) et OSGi (droite). Carzaniga

Dans le cadre de cette thèse nous suivons une démarche similaire à celle dans [Crnkovic 2006] et [Lau 2007], où sont proposés des modèles « idéalisés » du cycle de vie des composants, qui couvrent tout le spectre du développement logiciel, i.e. après la spécification des exigences (extra) fonctionnelles. Ainsi, le modèle comprend classiquement les étapes du cycle de vie qui vont de la **conception** et du **codage** à l'**exécution**, via la **compilation**, le **déploiement** et l'**activation** des composants.

3. A titre illustratif, la norme ISO/IEC 12207 [Singh 1995, IEEE 1998] qui traite sur le développement et la maintenance des logiciels, définit 23 « processus », 95 « activités », 325 « tâches » et 224 « résultats »

Nous définissons également des phases du développement logiciel, qui regroupent des étapes du cycle de vie dont les activités menées ont des caractéristiques communes. Ainsi, nous définissons les phases de **conception**, de **mise en œuvre** (regroupant la compilation et le déploiement) et d'**utilisation** des composants (regroupant l'activation et l'exécution). La figure 2.3 illustre ce modèle simplifié du cycle de vie des composants qui reste très général.

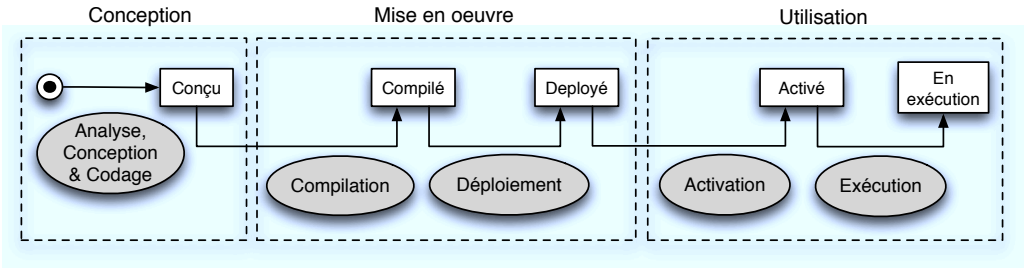


FIGURE 2.3 – Cycle de vie simplifié des composants. Les états du logiciel sont représentées comme des carrés et les étapes comme des ellipses. Les phases regroupent des étapes du cycle de vie.

Ce modèle nous est suffisant dans le contexte de la thèse pour positionner les modèles à composants et la génération de code associée. Par la suite, nous détaillons ces cinq étapes du cycle de vie pour les composants logiciels.

2.2.3.1 Conception et codage

La phase de conception regroupe les activités qui sont menées par des humains (éventuellement assistés par des logiciels spécialisés) afin de décrire et détailler progressivement les exigences fonctionnelles et extra-fonctionnelles du logiciel. On obtient ainsi une description du comportement désiré des logiciels, prenant en compte (si nécessaire) l'ensemble des contraintes et dépendances nécessaires vis-à-vis de l'environnement logiciel et matériel.

Dans le cas des logiciels à base de composants les retombées sont, d'un coté, une description *structurelle* présentant l'organisation des composants internes ainsi que leur connexions, et d'un autre coté une description *comportementale* exprimée en langages de programmation de haut niveau (C, C++, Java) ou des automates [4].

Compte tenu de l'importance de l'étape de conception du logiciel, elle constitue en elle seule une première phase du cycle de vie dont nous ne détaillons pas les multiples étapes internes.

2.2.3.2 Compilation

La compilation comporte les activités qui visent à traduire les descriptions structurelles et comportementales dans un langage « cible », qui est compréhensible par une machine physique, comme un processeur, ou une machine virtuelle.

4. Comme ça a été évoqué au chapitre 2.1.2.2, dans le cadre de cette thèse nous nous intéressons tout particulièrement aux composants dont leur comportement est décrit en langage C

Dans cette étape les concepts abstraits utilisés lors de la modélisation du logiciel à base de composants sont concrétisés en langage cible, tout en gardant la cohérence fonctionnelle du système. La traduction vers ce langage cible peut être directe, mais plus généralement elle implique l'activation de multiples compilateurs ou de passes de compilation succesives, ainsi que l'utilisation des langages intermédiaires.

2.2.3.3 Déploiement

Nous définissons le déploiement comme l'ensemble d'activités permettant le transfert ou la copie de composants à partir d'un noeud de **production** vers un ou plusieurs noeuds d'**exécution**, i.e. les plates-formes d'exécution. Cela comporte l'identification des ressources nécessaires pour l'exécution correcte des composants, le *packaging* des composants afin d'être transmis, la gestion de la délivrance de contenus et l'installation des composants [Carzaniga 1998].

Certaines des activités de déploiement peuvent être menées avant la compilation, comme lorsque le code source est compilé « à la volée » dans les plates-formes d'exécution ; dans ce cas la distribution de contenus se fait au niveau du code source. De façon similaire, des activités qui sont en lien direct avec les descriptions des sites de consommation, comme l'assignation de zones de mémoire physique à des symboles ou procédures, sont communément associées à la compilation, notamment s'agissant des systèmes embarqués.

Sur ce point particulier, dans le cadre de cette thèse nous considérons l'adressage des programmes comme une activité propre au déploiement.

Ensemble avec la compilation, ces deux étapes constituent la phase de mise en œuvre des composants, où les abstractions utilisées lors de la conception du système sont concrétisées en fonction des caractéristiques des plates-formes matérielles et logicielles sous-jacentes. La diversité des activités menées à cette phase répond à la grande hétérogénéité des plates-formes d'exécution. A la différence de la phase de conception, ces activités demandent une interaction humaine limitée à la configuration des outils de compilation et de déploiement.

2.2.3.4 Activation

L'activation d'un composant regroupe les activités nécessaires pour rendre ce composant « prêt à être exécuté » par d'autres composants voisins. Parmi celles-ci on trouve l'activation ou dé-activation de composants spécifiques et déjà présents sur la plate-forme, l'activation des composants dont le premier requiert des services (activation récursive), l'obtention d'un état stable d'exécution et le démarrage de processus annexes.

2.2.3.5 Exécution

Dans l'étape d'exécution le composant satisfait les exigences fonctionnelles et extra-fonctionnelles décrites dans la phase de conception, ce qui est évidemment l'objectif principal de tout développement. Au-delà de ces activités, nous incluons aussi dans cette étape des activités liées à l'évolution à l'exécution, telles que

la mise à jour, la maintenance et l'adaptation du logiciel, ainsi que les activités permettant aux composants de revenir à un état antérieur comme la désactivation et la désinstallation. L'évolution du logiciel sera l'objet du chapitre 4 de cette thèse.

Ces deux dernières étapes, l'activation et l'exécution, correspondent à la phase d'**utilisation** des composants logiciels, où ceux-ci, déjà déployés sur la plates-formes d'exécution, sont intégrés aux autres composants en exécution.

Nous avons défini ici, de façon très générale, les étapes du modèle de cycle de vie des composants, ainsi que les phases à très gros-grain qui se dérivent de cette modélisation. Dans le chapitre suivant nous nous intéressons à la manière dont les composants sont représentés et manipulés dans chacune de ces phases.

2.2.4 Les composants tout au long du cycle de vie

Afin de faciliter les activités menées dans chacune des étapes du cycle de vie, les composants sont souvent représentés des différentes manières. Par exemple, dans la phase de conception il ne semble pas adéquat de manipuler du code assembleur, qui est censé s'exécuter sur la plate-forme matérielle et logicielle. Chaque représentation évoque un niveau d'abstraction plus ou moins élevé du composant par rapport à un ou plusieurs aspects spécifiques du développement logiciel.

Notre intérêt à ce stade de l'étude est de montrer la façon dont les systèmes à base de composants sont construits et les éléments clés des langages utilisés pour les représenter tout au long des trois phases du cycle de vie identifiées au chapitre 2.2.3 : conception, mise en œuvre et utilisation.

Concernant la phase de **conception**, les modèles à composants fournissent un cadre de travail pour la construction d'une architecture logicielle qui satisfait les exigences des spécifications. Il indique les principes qui guident la conception des composants et la manière de connecter les composants entre eux. Il existe à ce jour une pléthore de modèles à composants qui sont utilisés dans des domaines très divers par des plates-formes de développement logiciel, dans le monde académique comme dans un contexte industriel. Rien que dans le domaine de l'embarqué on peut citer Koala [van Ommering 2000], Runes [Costa 2007], OpenCOM [Coulson 2008], CORBA [Hugues 2007], Rubus [Isovic 2002], Giotto [Horowitz 2003], AADL [Feiler 2004], LooCI [Hughes 2009], MMLite [Helander 1998], SOFA [Plásil 1998], CAmkES [Kuz 2007b], COMDES [Angelov 2006, Ke 2007], TECS [Azumi 2007], Think [Fassino 2002], SaveCCM [Hansson 2004], East-ADL [Cuenot 2007], Pecos [Genssler 2002] et TinyOS [Levis 2005]. Tous ces modèles définissent des abstractions similaires [Babau 2007], même s'ils exposent des particularités propres à leurs contextes d'application. Ces abstractions sont étudiées au chapitre 2.2.4.1.

Concernant les phases de **mise en œuvre** et d'**utilisation**, la grande hétérogénéité des plates-formes d'exécution et de sous-domaines d'applications rend difficile, voire impossible, une description homogène et unique la chaîne de développement associée à ces modèles à composants. C'est la raison pour laquelle l'étude sur ces phases est faite sur quatre des plates-formes de développement représentatives de

notre domaine d'étude, au chapitre 2.2.4.2.

2.2.4.1 Conception : les modèles à composants

L'objectif de ce chapitre est de présenter les principaux concepts qui sont partagés par la plupart des modèles : attributs, interfaces, connecteurs, contenu et assemblages de composants [Babau 2007]. Selon le modèle choisi, ces abstractions peuvent avoir des noms différentes.

Attributs

Les attributs représentent les propriétés structurelles, comportementales, fonctionnelles ou extra-fonctionnelles des composants. Même si l'idée d'attribut dans le contexte des modèles de composant est très intuitive, l'univers très large et la grande variété de propriétés des composants auxquelles on peut faire référence rend difficile la formalisation du terme. La valeur d'un attribut peut être liée plus ou moins étroitement à l'implémentation du composant, à son interaction avec les autres composants, ou au fait d'avoir implanté une interface particulière.

En se basant sur les travaux de Shaw [Shaw 1996a], Sentilles et. al. [Sentilles 2009] propose une définition que nous adoptons dans le cadre de cette thèse :

$$\begin{gathered} attribute =< IdType; Valeur+ > \\ Valeur =< Donnée; Métadonnée; ConditionsDeValidité > \end{gathered} \tag{2.1}$$

Où :

- *IdType* désigne la classe d'attribut : le nom de la classe, le format utilisé pour représenter l'attribut et les entités du modèle sur lesquelles l'attribut peut s'appliquer.
- *Donnée* contient la valeur concrète de l'attribut.
- *Métadonnée* fournit des information supplémentaires sur la valeur de l'attribut.
- *ConditionsDeValidité* définit les conditions de validité de l'attribut.

  Les travaux de Sentilles considèrent que les attributs sont applicables non seulement aux composants mais aussi aux interfaces, connecteurs et toute autre entité du modèle de composant. Afin de simplifier la présentation, dans le cadre de cette thèse nous considéreront les attributs comme propres seulement aux composants.

Interfaces

Les interfaces sont les seuls points d'interaction d'un composant avec son environnement. Elles représentent la spécification du composant, puisque idéalement toutes les dépendances de contexte doivent être capturées par les interfaces de celui-ci.

On trouve des interfaces en entrée ou en sortie, utilisées lorsque la communication entre composants est de type flot de données ou événementielle comme dans Giotto [Horowitz 2003] et Rubus [Isovic 2002], et des interfaces fournies et requises, utilisées lors des communications du type client/serveur et dans des architectures orientées services comme dans Think [Fassino 2002, Anne 2009], Koala [van Ommering 2000], CAmkES [Kuz 2007b] ou UML2.0 [Pilone 2005] Dans le cadre de cette étude nous nous focalisons sur ce dernier type d'interfaces.

Un composant qui expose une interface requise demande un ensemble de services définis par la description de l'interface ; un composant qui offre une interface fournie assure que son comportement correspond à ce qui est décrit par l'interface si ces interfaces requises sont correctement connectées. La description d'une interface peut être plus ou moins explicite suivant la richesse du contrat qui peut être établi entre deux interfaces du même type. Dans [Beugnard 1999, Beugnard 2010], Beugnard et. al. définissent quatre types de contrats progressivement plus riches : basiques ou syntaxiques, comportementaux, de synchronisation et quantitatifs.

Dans cette étude, nous nous concentrons sur les interfaces qui établissent des contrats syntaxiques. Dans ce cas, la description d'une interface est la liste des signatures de ses méthodes : leur nom, les types des paramètres de retour et les noms et types des paramètres d'entrée. Ces informations sont exprimées dans un langage dédié à la description d'interfaces, ou IDL.

Connecteurs

Les connecteurs modélisent les interactions entre les composants et réalisent l'établissement des contrats. Ils spécifient les règles qui régissent la relation entre deux ou plusieurs composants via leurs interfaces, ainsi que la politique de communication (« pipe », mémoire partagée, rendez-vous, section critique...).

La complexité des connecteurs varie selon les approches [Mehta 2000]. Ils peuvent être modélisés par des composants dotées d'interfaces qui assurent le lien entre des interfaces hétérogènes, ou par des simples liens logiques entre interfaces du même type. Un certain nombre des modèles à composants permettent les deux types de modélisation. Dans cette étude nous considérons les connecteurs basiques, ici des appels de procédure, adaptés aux modèles à composants de l´embarqué de type client/serveur en C.

Contenu

Le contenu d'un composant est associé à la description de son comportement. Contrairement aux interfaces et aux attributs, qui font référence aux propriétés observables ou phénotype du composant, le contenu fait référence aux propriétés internes et donc non observables dans une approche "boîte noire". Ainsi, alors que les interfaces décrivent quels services sont offerts ou requis, l'implémentation décrit comment ces services sont assurés.

A partir des deux manières de décrire le contenu on peut distinguer deux types de composants :

– Pour les composants du type **composite**, le contenu est décrit par un assemblage de sous-composants et par les connexions entre ceux-ci. L'assemblage de composants fera l'objet du paragraphe suivant.

– Pour les composants du type **primitif**, le contenu est décrit à l'aide d'un langage "classique" comme le C, le C++ ou Java [Gosling 2000], ou d'un langage formel comme ceux basés sur des automates à états finis. Comme cela a été établi au chapitre 2.1.2.2, nous nous intéressons aux composants dont le comportement est décrit dans le langage C.

Assemblage

Les composants sont assemblés en satisfaisant les services requis d'un composant grâce aux services fournis par un autre composant, les liant à travers d'un connecteur. Le plus couramment, un assemblage de composants [5] est étroitement lié à l'existence d'un composant du type composite, qui à son tour peut faire partie d'un autre assemblage. Cela permet de créer une hiérarchie de composants et, au sens large, de construire des architectures logicielles, définies par Shaw et Garlan [Shaw 1996b] ainsi :

« Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns ».

Dans la plupart des modèles à composants le terme architecture est en fait associé à des assemblages de composants. Les architectures sont le plus souvent décrites dans un langage dédié ou ADL. De nombreux ADL ont été proposés tant pour des domaines spécifiques comme pour des fins plus générales. Il y a peu de consensus sur ce qu'est un ADL et quel rôle doit-il jouer dans la modélisation d'une architecture : pour certains il doit avant tout faciliter la compréhension et la communication sur les systèmes logiciels, privilégiant la simplicité et des représentations graphiques. Pour d'autres, un ADL doit posséder une syntaxe et une sémantique formelles, ainsi que des outils d'analyse et de vérification. Pour une comparaison de langages de description architecturales nous conseillons la lecture des travaux de Medvidovic et Taylor [Medvidovic 2000].

2.2.4.2 Mise en œuvre et utilisation : 4 plates-formes représentatives du domaine

L'objectif final de tout processus de développement à base de composants est le déploiement des systèmes représentés par les abstractions précédemment décrites dans une plate-forme matérielle et/ou logicielle, et son exécution. Compte tenu de

5. Dans la définition même du terme composant on trouve la notion de composition, entendue comme l'association de plusieurs éléments qui forment un tout. Dans cette thèse nous préférons utiliser le terme assemblage, en raison du sens ambigu du terme composition, que certains ne lient qu'à la présence de plusieurs niveaux de hiérarchie dans un composant.

l'hétérogénéité des plates-formes d'exécution et de sous-domaines d'applications évoquée précédemment, nous avons sélectionné quatre plates-formes de développement à base de composants logiciels qui nous semblent représentatives du domaine. Nous privilégions dans cette sélection les modèles natifs, i.e. ceux où le comportement des composants logiciels est décrit dans un langage de programmation qui est ensuite compilé est exécute par un processeur, comme le langage C. Cela exclut des modèles à composants tels que .NET [Wigley 2002], Fractal-Julia [Bruneton 2004a], iPojo [Escoffier 2007] ou EJB [Burke 2006], où la plate-forme minimale d'exécution ou *runtime* est trop coûteuse par rapport aux ressources physiques disponibles dans la plupart des cas.

Dans cette étude nous avons retenu les plates-formes **TinyOS** [Levis 2005, TinyOS 2010], **CamkES** [Kuz 2007b, CAmkES 2010], **Koala** [van Ommering 2000, van Ommering 2002] et **Fractal/Think** [Bruneton 2004a, Fractal 2010]. Par la suite, pour chacune d'entre elles nous proposons :

- une présentation générale de la plate-forme et du type d'application qu'elle adresse,
- une description de l'outillage fourni pour la conception de systèmes et des particularités par rapport aux abstractions communes définies au chapitre 2.2.4.1,
- une description de la mise en œuvre (compilation et déploiement) des composants, tant au niveau des activités effectuées comme de l'outillage utilisé, et
- une description de l'utilisation des composants et des aspects à considérer lors de l'exécution de ceux-ci.

TinyOS

Présentation. TinyOS [Levis 2005, TinyOS 2010] se définit comme un système d'exploitation (OS) adapté aux applications du type réseaux de capteurs (WSN) et conçu en fonction des exigences de ce domaine : ressources limitées, exécution d'applications concurrentes et basse consommation d'énergie. Pourtant, TinyOS n'est pas tout à fait un OS au sens traditionnel, il est plutôt un environnement de programmation à base de composants, permettant de construire des applications et des OS adaptés aux besoins spécifiques des applications. Il est conçu et soutenu par l'université de Berkeley en Californie (US).

Conception. Un système dans TinyOS est un graphe de composants. Il en existe de deux types : des modules, qui décrivent un comportement lié à une ou plusieurs interfaces (et sont donc assimilables aux composants primitifs) et des configurations, qui contiennent des sous-composants et les connexions que les lient entre eux (assimilables aux composants composites).

Les interfaces sont orientées et bidirectionnelles. Au sein des interfaces on considère deux types de méthodes, suivant le sens des appels : des commandes, du client vers le serveur, et les événements, du serveur vers le client. Les commandes repré-

sentent un appel à un service, dont la réponse et signalée par un événement. Les événements sont aussi utilisés pour modéliser les interruptions du matériel.

Le comportement des composants, ainsi que toutes les artefacts du modèle, sont décrits dans un langage dédié, nesC [Gay 2003], un dialecte du langage C qui impose deux limitations par rapport aux implémentations : il interdit les pointeurs sur fonction et ne supporte pas l'allocation dynamique de mémoire. Selon les auteurs, ces limitations ne sont pas coûteuses en terme pratique. Par contre, l'utilisation d'un tel langage ad-hoc limite l'intégration de code *legacy* dans les applications, un besoin récurrent dans le domaine des systèmes embarqués.

L'assemblage des composants est aussi décrit en langage nesC. De plus, un seul fichier source regroupe la description des services offerts et requis par le module (une vision « boite noire » du composant) et le comportement associé à ces services (une vision « boite blanche »). Cela ne facilite pas la création d'un dépôt de modules et de configurations, ni la réutilisation de composants.

Mise en œuvre. La compilation d'un système basée sur des composants TinyOS est mené par nescc, un compilateur pour le langage nesC qui étend le front-end de la version 2.8.1 de GCC [FSF 2010b] afin de prendre en compte les particularités du langage. Elle se fait en quatre étapes [nesC 2010] :

- *Chargement* : dans un premier temps, nescc charge la description des composants (des fichiers .nc) et construit des noeuds AST (Abstract Syntax Tree) de leur code source, le comportement des composants inclus. Il exécute également un ensemble de vérifications sémantiques liées au langage nesC : résolution de symboles, vérification de types et d´utilisation d´instructions du type `break`, `goto` ou `continue`, etc. Ce processus se fait de manière récursive à partir du fichier .nc du composant racine.
- *Instanciation et connexion* : les composants génériques sont instanciés et un graphe des connexions de l'application est construit.
- *Propagation de constantes* : les expressions qui sont évaluées comme constantes à la compilation sont remplacées par la valeur correspondante.
- *Génération de code* : pour chaque instance de composant, nescc génère un fichier C qui contient le code des modules et les fonctions de liaison qui correspondent aux connexions définies par les configurations. Il applique aussi des optimisations telles que l'*inline* des fonctions ou l'enlèvement des fonctions non invoquées. Ces optimisations sont détaillées au chapitre 3.6.2 de cette thèse.

Le code C généré par nescc est ensuite compilé en binaire exécutable par un compilateur classique. Des cross-compilateurs pour les microcontrôleurs AVR et MSP430 sont actuellement supportés.

En principe, une application à base de composants TinyOS n'exige pas une plate-forme logicielle sous-jacente et peut être directement déployée sur une plate-forme matérielle. Dans la pratique des composants du type OS forment une couche logicielle entre l'application et le matériel. La distribution officielle de TinyOS fourni une bibliothèque de composants encapsulant des services système (ordonnanceur, pools de mémoire, queues...) et des services liées aux plate-formes matérielles (pilotes de périphériques, gestion de protocoles série, piles de communi-

cation radio...). Cette bibliothèque est portée aux plate-formes MICA [Hill 2002] et TMoteSky [Moteiv 2006] conçues en fonction des besoins des réseaux de capteurs.

Utilisation. Les caractéristiques du langage nesC, notamment le découpage des méthodes des interfaces en événements et en commandes, incitent l'utilisation du motif appel/réponse ou split-phase. Afin de gérer l'exécution des événements/commandes, les développeurs de TinyOS ont choisi une approche similaire aux messages actifs [von Eicken 1992, Buonadonna 2001]. Cela signifie qu'un nouvel événement/-commande préemptera l'événement/commande en cours d´exécution.

De plus, TinyOS dispose d'un mécanisme de tâches qui s'exécutent en mode *run-to-completion* et ne sont donc pas préemptibles par une autre tâche. L'exécution d'une tâche est suspendue lors de l'arrivée d'un événement ou de l'invocation d'une commande. Les tâches sont utilisées dans des étapes de calcul pouvant être réalisés comme tâches de fond. En remplaçant l'ordonnanceur de la bibliothèque de composants il est possible d'implémenter d'autres politiques de gestion de tâches.

En somme, TinyOS impose un modèle d'exécution et de communication entre tâches bien adapté au domaine des réseaux de capteurs, mais assez coercitif lorsqu'on se situe dans un autre domaine de l'embarqué.

CAmkES

Présentation. Le projet CAmkES [Kuz 2007b, CAmkES 2010], de l'anglais *Component Architecture for micro-kernel based Embedded Systems*, s'inscrit parmi les initiatives du centre d´excellence en technologies de l'information australien, NICTA. Il propose un modèle à composants flexible, extensible et à faible surcoût en occupation mémoire, destinée à être utilisé pour concevoir des services OS (pilotes de périphériques, systèmes de fichiers...) et des applications exécutant sur un micro-noyau.

Conception. Tout comme TinyOS, CAmkES supporte des composants primitifs et composites. Les composants primitifs sont implémentés en langage C avec des règles particulières concernant les noms des méthodes, qui doivent être préfixés par le nom de l'interface auquel il fait référence. Aux deux types de composants on peut associer une configuration, i.e. un ensemble d'attributs dont les valeurs représentent le statut du composant ou ses paramètres de réglage. Tous les composants sont décrits dans un ADL ad-hoc.

Le modèle CAmkES supporte trois types d'interface : des appels standards à des procédures, des événements (modèle publish/subscribe) et des *dataports* qui représentent des variables partagées entre composants. Les types des interfaces sont décrits dans un IDL ad-hoc, inspiré du CORBA IDL [OMG 2010].

Dans CAmkES, les liaisons entre composants sont des structures de premier niveau, typées et implémentées de manière similaire aux composants. Un connecteur définit l'ensemble de types d'interfaces qui peuvent être liées par celui-ci, ainsi que les mécanismes de connexion ; une connexion est une instance d'un connecteur

dans une architecture. Cette distinction permet d'abstraire les aspects liés à la communication entre composants qui peut être faite de diverses manières, via des sémaphores, des mutex, etc.

Mise en œuvre. La compilation d'une application CAmkES se fait en deux étapes. Dans un premier temps, un squelette du code d´implémentation C est généré à partir des fichiers ADL et IDL. Notamment, en fonction des types de connexions instanciés dans l'architecture, le compilateur génère les entêtes des fonctions correspondantes. Par exemple, dans le cas où une connexion du type *publish/subscribe* est instanciée, une fonction `{interfaceName}_wait(void)` est crée. Cette étape est considérée comme une aide à la conception de l'application.

Une fois le comportement des composants est décrit, le compilateur CAmkES (écrit en Python [Python 2010]) génère du code « glue » qui implémente la communication entre les composants. Dans le cas des appels à procédures des macros (`#define`) sont créées. Pour les connecteurs d´une complexité supérieure, des *stubs* générique, à être développés à l'avance, sont chargés et adaptés à la structure de l'application ; par exemple, une connexion entre deux composants peut être réifiée par une boîte aux lettres liée à ces composants. Le compilateur genere également du code d´instanciation et d'initialisation des instances des composants. Une chaîne de compilation classique (GCC) est ensuite utilisée pour transformer ce code en binaire exécutable.

Dans l´état actuel, les composants CAmkES, et en particulier le code généré par le compilateur, sont fortement liés à une plate-forme logicielle particulière. Elle est composée du micro-noyau L4 [Liedtke 1995] qui fournit les fonctionnalités minimales d'un système d'exploitation s´exécutant en mode privilégié du processeur (mapping de mémoire entre espaces d´adressage, primitives de communication inter-processus) et du superviseur d'OS Iguana [Heiser 2005], qui s´exécute en mode utilisateur et qui fourni les fonctionnalités que l'on pourrait attendre d'un système d'exploitation classique : gestion de la mémoire, gestion de la protection (via un processus indépendant qui est associé à chaque composant par défaut), des mécanismes de partage de données et de communication entre processus.

En conséquence, les entités du modèle CAmkES sont transformées à la compilation en abstractions propres à Iguana : les composants sont transformés en serveurs s´exécutant dans des différentes domaines de protection, les dataports en régions de mémoire partagée, les événements en notifications, et ainsi de suite.

Utilisation. Le lien très fort qui existe entre la chaîne de développement CAmkES et la plate-forme logicielle sous-jacente impose un certain nombre d´inconvénients concernant l´exécution des composants. Tout d'abord, les composants CAmkES, puisqu'ils sont implémentés comme serveurs Iguana, sont associés à des processus qui ont des domaines de protection de mémoire indépendants les uns des autres. La gestion de la mémoire, définie par la plate-forme logicielle sous-jacente, exige des plate-formes matérielles la présence d'une unité de protection de mémoire (MPU) ou idéalement une unité de gestion de mémoire (MMU). L'activation des composants CAmkES est également dépendante des politiques d'activation des serveurs Iguana.

Le micro-noyau L4 n´est supporté que par des processeurs à 32 et 64 bits. Également, l'espace mémoire consommé par la plate-forme logicielle n'est pas négligeable : 200KB pour le micro-noyau L4 et 100KB pour Iguana, pour un processeur de la famille ARM. Ces exigences vis-à-vis de la plate-forme matérielle de la part de la plate-forme logicielle minimale empêchent l'utilisation de cette chaîne de développement dans le cadre des systèmes embarqués très contraints.

Koala

Présentation. Koala [van Ommering 2000, van Ommering 2002] est un modèle à composants conçu au sein de la société Philips au Pays-Bas, afin d'encourager la réutilisation de logiciel embarqué dans le domaine de l´électronique grand public. Il vise à gérer la complexité croissante des applications, ainsi que la grande diversité entre familles de produits, tout en prenant compte des limitations au niveau de ressources physiques.

Conception. Un composant Koala est une entité de conception architecturale qui communique avec son environnement au travers de ses interfaces. Le modèle fait la distinction entre le type du composant, i.e. un composant réutilisable, et une instance, i.e. une occurrence d'un tel composant dans une *configuration*. Une configuration est un composant qui contient des sous-composants et des connexions entre leurs interfaces (assimilable donc au composite). Les composants sont décrits dans un ADL propriétaire (appelé aussi Koala). Le comportement des composants est exprimé en langage C avec des règles similaires à celles de CAmkES concernant les noms des fonctions.

Les interfaces sont décrites à l'aide d'un IDL propriétaire inspiré des interfaces COM et Java. Elles contiennent la liste des signatures des fonctions. Une "interface de diversité" est une interface cliente dont les méthodes permettent d´accéder et de définir les propriétés du composant. Les interfaces de diversité sont utilisées pour configurer le comportement et l´architecture interne du composant : les valeurs des propriétés du composant ne sont pas remplies par un outil de configuration ou définies lors de l'instanciation de celui-ci, mais plutôt requises par le composant et définies par d'autres composants en dehors de la configuration (cf. figure 2.4).

Concernant les connecteurs, on en distingue trois types : des appels de procédure classiques, des modules qui modélisent des opérations plus complexes concernant une interface cliente et un nombre limité d´interfaces serveur, et des *switches*. Ces derniers représentent une connexion qui peut être modifiée dynamiquement et qui lie une interface cliente à une seule interface serveur parmi les n interfaces qui sont liées aux *switch*. La « position » du *switch*, i.e. laquelle des interfaces serveur est effectivement connectée à l'interface cliente, est contrôlée par une interface de diversité. La figure 2.4 illustre une configuration où l´interface de diversité d´un *switch* détermine quel des deux composants `B` ou `C` sera invoqué lorsque le composant `A` demande le service représenté par l'interface `Ia`.

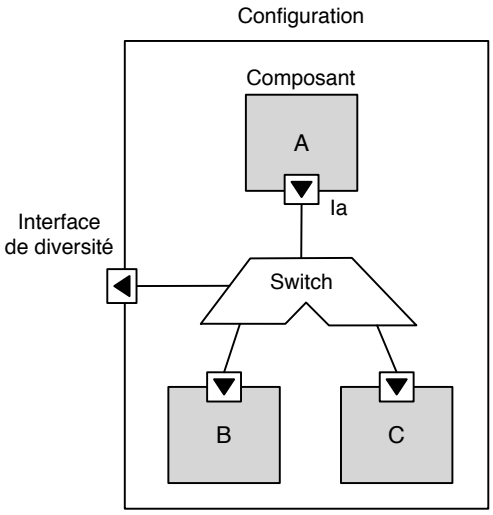


FIGURE 2.4 – Une configuration Koala contenant trois composants et un switch côntrolé via une interface de diversité

Mise en œuvre. Le compilateur Koala génère du code « glue » qui est composé majoritairement de directives du type `#define` qui effectuent :

- Le renommage des fonctions dans le code d´implémentation, afin d'assurer la correspondance, pour chaque connecteur, entre les fonctions appelées du coté client et les fonctions existantes du coté serveur.
- La gestion des multiples instances d'un composant. Le compilateur Koala ne manipule pas le code source fourni par le développeur. Par conséquent, celui-ci doit prévoir qu´un composant est multi-instanciable et rajouter un premier paramètre aux méthodes des interfaces serveur qui est en fait un pointeur vers une structure de données uniques au composant. Pour les composants clients, Koala génère des macros afin de rajouter un pointeur à cette structure de données comme premier paramètre lors des appels de fonctions.

Le compilateur effectue aussi des simplifications qui concernent les interfaces de diversité et les *switches*. Par rapport aux premières, lorsque les propriétés auquel l'interface de diversité fait référence, possède des valeurs constantes à la compilation, les appels aux fonctions sont remplacés par ces valeurs. Concernant les *switches*, puisque c'est une interface de diversité qui contrôle la « position » de celui-ci, lorsque la propriété est constante, le *switch* est implémenté comme un appel de procédure classique. Nous détaillerons ces aspects au chapitre 3.6.4 de cette thèse.

Le code fourni par le développeur et le code « glue » généré par Koala sont transformés en exécutables binaires par des compilateurs classiques. Puisque le compilateur Koala n'est pas un logiciel libre, nous ne possédons pas plus d'information concernant cette deuxième phase de compilation.

Utilisation. Koala ne fait aucune hypothèse concernant les plates-formes lo-

gicielle et matérielles sous-jacentes ou les modèles d´exécution. Par rapport à ces derniers, des patrons de conception sont proposés afin d´implémenter des événements ou de tâches.

Fractal/Think

Présentation. Fractal [Bruneton 2004a, Fractal 2010] est un modèle à composants conçu par France Telecom R&D et l´Institut National de Recherche en Informatique et Automatique (INRIA), en France. Il vise l'administration de systèmes logiciels et offre pour cela des mécanismes de structuration du code et des données, ainsi que des mécanismes optionnels de contrôle qui déterminent le niveau de conformité des composants par rapport au modèle. Think [Fassino 2002, Anne 2009, ThinkTeam 2010] est une des implémentations du modèle Fractal qui vise le développement des systèmes embarqués.

Conception. Un composant Fractal est constitué d'un contenu, d'une membrane et d'un ensemble d'interfaces serveurs et/ou clientes. Tant le contenu que la membrane peuvent contenir soit les implémentations des interfaces serveur, soit des sous-composants. La différence entre les deux est subtile : le contenu ne concerne que les aspects fonctionnels du composant, alors que la membrane concerne des aspects extra-fonctionnels comme la gestion du cycle de vie ou la réflexivité. Cette distinction permet la séparation nette entre préoccupations fonctionnelles et celles liées à l'administration des composants à l´exécution. Comme Koala, Fractal fait la distinction entre une définition de composant (son type) et une instance de composant.

La spécification Fractal [Bruneton 2004b] décrit les entités du modèle de manière très générique, un nombre de détails nécessaires à une utilisation concrète du modèle sont laissés ouverts. Par exemple, Fractal n'impose pas l'utilisation d'un langage de programmation en particulier pour l´implémentation des contenus et des membranes. Fractal est un cadre de définition des modèles à composants (une sorte de méta-modèle) qui doit être « instancié » par une implémentation plus concrète.

Think est un environnement de développement qui concrétise les concepts du modèle à composants Fractal dans le domaine de l'embarqué. Il introduit un certain nombre de différences, comme le fait que les composants composites peuvent aussi être associés à un comportement décrit dans un langage de programmation classique.

Les contenus et les membranes des composants Think sont implémentés en langage C. Les interfaces sont décrites dans un IDL ad-hoc qui liste les signatures des méthodes correspondantes. Les architectures sont décrites dans un ADL ad-hoc. La sémantique des liaisons (i.e. les connecteurs) entre composants n'est pas explicitement définie par Think, elle correspond par défaut à des appels de procédure, mais via des extensions à la chaîne de compilation un connecteur peut avoir un comportement plus complexe et être même implémenté comme un composite ou des composants primitifs enchaînés, comme dans [Polakovic 2008, Poulhies 2010].

Mise en œuvre. Le compilateur Think, écrit en Java, traite les fichiers ADL, IDL et C en deux étapes :

- *Chargement* : un AST est construit à partir des descriptions des composants et des interfaces. Chaque noeud de cette AST représente une entité du modèle à composants. Ensuite, une série enchaînée de modules de chargement exécutent des validations sémantiques sur l'architecture de l'application (e.g., vérifier qu'un connecteur lie une interface cliente à une interface serveur et non à l'inverse) et des modifications de l'AST (e.g., l'inclusion dans l'application de composants chargés d'une bibliothèque, de manière automatique).
- *Réécriture et génération de code* : le code C fourni par le développeur est réécrit afin d'introduire les concepts propres au modèle à composants. Les transformations concernent principalement le renommage des fonctions, des attributs et des données privées à l'instance du composant, afin de leur donner un nom unique dans l'application. De plus, pour chaque instance de composant un ensemble de méta-données sont générées par la chaîne de compilation. Ces manipulations utilisent une représentation abstraite du code C qui est obtenue à l'aide d'un paquet logiciel associé, Codegen.

Le compilateur Think génère un ensemble de fichiers en langage C qui sont ensuite traités par des compilateurs classiques comme GCC ou IAR [IAR 2010].

Contrairement à Koala, les composants Think sont par défaut multi-instanciables. Il est donc possible de créer dynamiquement une instance d'un type de composant, pourvu qu'un composant dédié du type « Factory » soit aussi déployé. De même, toutes les liaisons entre composants et les accès aux attributs et aux données privées sont dynamiques, i.e. elles comportent un ou plusieurs niveau d'indirection. L´implémentation des liaisons en Think ressemble à celle des objets dans des langages tels que C++.

Utilisation. Comme Koala, Think ne fait aucune hypothèse concernant les plate-formes logicielle et matérielles sous-jacentes ou les modèles d´exécution. La dynamicité inhérente des applications Fractal/Think, dans le sens où elles sont construites pour pouvoir être modifiées en structure et comportement après leur déploiement, introduit des surcoûts à payer en termes de consommation de ressources matérielles qui seront étudiés au chapitre 4.4.1.3.

2.2.4.3 Synthèse sur les composants tout au long du cycle de vie

A partir de l'étude des modèles à composants, et en particulier de ceux présentés au chapitre 2.2.4.2, malgré les dénominations distinctes, les abstractions utilisées dans la conception de systèmes basés sur ces modèles à composants se retrouvent dans les notions suivantes, détaillées au chapitre 2.2.4.1 : les attributs représentent des propriétés typées, les interfaces fournies et requises offrent des points d'interaction, les connecteurs permettent de lier des interfaces requises à des interfaces fournies, le contenu décrit le composant en langage C, et les assemblages de composants permettent de construire des systèmes. Cette relative homogénéité a pour cause la très faible dépendance des systèmes vis-à-vis les plates-formes d'execution pendant la phase de conception de ceux-ci.

De leur côté, les activités des phases de mise en œuvre et d'utilisation sont progressivement plus dépendantes des plates-formes d'exécution sous-jacentes. La grande hétérogénéité existante au niveau des plates-formes matérielles et logicielles rend difficile une analyse unique des chaînes de développement associées. A partir de l'étude des modèles à composants, nous identifions quatre éléments principaux qui déterminent la mise en œuvre et l´exécution des composants logiciels :

- **Les plates-formes logicielles sous-jacentes**. La mise en œuvre des composants est influencée par l'hypothèse d'une plate-forme logicielle. Les services offerts par cette couche logicielle déterminent les politiques de compilation et de déploiement. Nous distinguons à ce sujet quatre familles :
  - Les machines virtuelles, où la compilation des composants a comme résultat un bytecode destiné à être interprété par celle-ci, et non du code binaire exécutable par un processeur. Cette famille va des machines virtuelles plus génériques comme celles basées sur Java [AonixPERC 2010], à celles adaptées à un domaine très spécifique comme Maté [Levis 2002] (basée sur TinyOS), DAViM [Michiels 2006] ou Tapper [Xie 2006] pour les réseaux des capteurs. Par les considérations évoquées précédemment, nous n'avons pas étudié des modèles à composants basés sur des machines virtuelles dans le cadre de cette thèse.
  - Les systèmes d´exploitation (OS), où la compilation des composants et notamment la génération de code sont fortement attachées aux fonctionnalités offertes par un OS spécifique. Tel est le cas de CAmkES, où les abstractions du modèle à composant sont transformées en abstractions Iguana et L4.
  - Les intergiciels d´exécution, à mi-chemin entre les deux précédentes. Dans cette famille une couche logicielle s´occupe de certains aspects communes aux composants d´une application. Dans RUNES [Costa 2007] des Capsules gèrent les aspects liés à l´adaptabilité via la réconfiguration dynamique des composants qu´elles regroupent. Dans Fractal/Think les membranes des composants gèrent les aspects extra-fonctionnels des composants ; parmi ces aspects on trouve également l´adaptabilité du logiciel à l'exécution.
  - La non explicitation de plate-forme logicielle sous-jacente comme dans Koala, où aucune hypothèse est faite par rapport à celles-ci au moment de la compilation et le déploiement des composants.
- **Les plates-formes matérielles visées**. Les chaînes de compilation de logiciels à base de composants sont souvent conçues pour un domaine d'application spécifique et supportent un nombre limité d'architectures matérielles ou de familles de processeurs. CAmkES, par exemple, ne peut être utilisé que dans le cas où la plate-forme matérielle contient une unité de gestion de la mémoire (MMU).
- **Les modèles d´exécution**. Certains modèles à composants imposent ou incitent l'utilisation d'un modèle d´exécution donné (e.g. basé sur des tâches ou sur des événements). Ceci impacte la génération de code et requiert l´utilisation de plates-formes logicielles spécifiques.
- **Les compilateurs de bas niveau**. Les chaînes de compilation de logiciels à base de composants font souvent appel à des chaînes de compilation existantes

afin d'obtenir un code binaire exécutable. Par conséquent, elles sont soumises aux restrictions du compilateur choisi.

En règle générale, parmi le plates-formes de développement étudiées dans ce chapitre, le comportement des composants qui est fourni par les développeurs, i.e. le code source qui décrit le comportement des composants primitifs et les architectures logicielles des composites, est modifié par les chaînes de compilation, afin de l'adapter aux plate-formes logicielles et matérielles sous-jacentes, aux modèles d´exécution fixés par les modèles et aux outils de compilation de bas niveau. Également, les compilateurs sont souvent amenés à générer du code « glue », i.e. du code source qui implémente des fonctionnalités transparentes au développeur assurant l'interopérabilité entre les composants conçus de manière indépendante par les développeurs.

2.3 Synthèse et cadre de la thèse par rapport au domaine

Le lien très fort entre le logiciel et les processus physiques a comme résultat une grande hétérogénéité d'applications dans le domaine des systèmes embarqués. Compte tenu des considérations concernant les tendances de l'industrie du logiciel embarqué, présentées à l'introduction de cette thèse, nous nous focalisons sur des systèmes à un **haut dégré d'interaction** avec des agents externes, très **ouverts aux changements** et avec des **fortes contraintes** au niveau de l'utilisation de ressources physiques. La sûreté de fonctionnement, quoiqu'essentielle dans le domaine, n'est pas traitée au cours de cette thèse.

A partir de l'étude réalisée sur les modèles à composants et les plates-formes de développement associées, au chapitre 2.2.4, nous remarquons qu'il existe à ce jour des chaînes de développement logiciel qui intègrent les bénéfices des modèles à composants pour la conception des systèmes embarqués visés. Chacune possède des caractéristiques qui la rendent particulièrement adaptée à un domaine d'application des systèmes embarqués et qui limitent son utilisation à d'autres. Parmi les points communs, on constate qu'elles génèrent du code source C à partir des descriptions des composants et qu'un compilateur classique tel GCC est ensuite invoqué afin de effectuer la transformation dans un binaire exécutable.

Nous remarquons que, parmi ces chaînes de développement, certaines mettent l'accent sur l'optimisation du comportement des composants et de la « glue » éventuellement générée par les compilateurs (TinyOS et Koala). En ce qui concerne l'ouverture aux changements, nous soulignons la dynamicité inhérente des applications Fractal/Think, où toutes les entités du modèle à composants sont concrétisées de façon à faciliter sa modification après déploiement. Cette caractéristique la rend particulièrement adaptée à l'administration et la maintenance des composants à l´exécution.

Nous remarquons pourtant qu'aucune des plates-formes de développement existantes alloue un poids égal à ces deux préoccupations : l´utilisation raisonnée des ressources physiques et l'ouverture aux changements.

Après ces considérations d'ordre général, aux deux chapitres suivants nous étudions plus en détail les techniques d'optimisation logicielle qui permettent de satis-

faire les exigences au niveau de la consommation de ressources physiques (chapitre 3), et ensuite l´évolution logicielle, qui englobe tous les aspects liés au changement du logiciel (chapitre 4). Les modèles à composants décrits au chapitre 2.2.4.2 seront revisités dans ces chapitres afin de mieux évaluer leur performance vis-à-vis de ces deux aspects.

Chapitre 3

Optimisation

Sommaire

3.1 Définition **35**
3.2 Critères de performance **36**
3.2.1 Occupation mémoire 36
3.2.2 Temps d'exécution 36
3.2.3 Consommation d'énergie 37
3.3 Compromis **37**
3.4 La performance des composants logiciels et leur optimisation **38**
3.5 Optimisation des performances en fonction du cycle de vie **39**
3.5.1 Conception 40
3.5.2 Mise en œuvre 40
3.5.3 Exécution 41
3.6 Optimisation dans les plates-formes de développement de référence **42**
3.6.1 Préambule sur l'impact de la compilation de bas niveau . . . 42
3.6.2 TinyOS 43
3.6.3 CAmkES 44
3.6.4 Koala 45
3.6.5 Fractal/Think 45
3.7 Synthèse sur l'optimisation **46**

L'immaturité de l'ingénierie logiciel par rapport à d'autres branches scientifiques, et notamment celle des modèles de calcul par rapport aux modèles analytiques basés sur des équations mathématiques [Henzinger 2006], nous amène à redéfinir le concept d'optimisation. Comparée à l'optimisation au sens mathématique, l'optimisation du logiciel ne cherche pas à trouver *le* seul et unique ensemble de valeurs qui définissent une performance optimale, mais cherche plutôt à se rapprocher le plus possible de ce point d'optimalité, s'il existe.

Dans ce chapitre, nous définissons ce que l'optimisation du logiciel signifie dans le cadre de cette thèse et nous présentons les critères de performance qui nous intéressent, ainsi que les facteurs qui déterminent la performance des composants logiciels.

3.1 Définition

Nous définissons l'optimisation du logiciel comme la modification d'un système logiciel visant l'amélioration de sa performance à l'exécution par rapport à un ou plusieurs critères, tout en gardant sa cohérence sémantique.

A partir de cette définition assez générale nous proposons trois questions dont les réponses nous permettront de cadrer notre notion d'optimisation par rapport à notre domaine d'étude, et qui seront l'objet des paragraphes suivants :

- Quels sont les **critères de performance** utilisés lorsqu'on optimise le logiciel ?
- Quels sont les **compromis** à faire entre les critères de performance ?
- Quels sont les **facteurs** qui déterminent la performance d'un composant ?

3.2 Critères de performance

La recherche de l'efficacité dans l'utilisation des ressources disponibles est un enjeu majeur du développement logiciel, particulièrement dans l'embarqué. Elle a donc été l'objet des nombreux travaux de recherche. Nous identifions trois des critères de performance sur lesquels ces efforts se sont concentrés que nous jugeons comme les plus représentatifs dans le cadre de cette thèse, à savoir : l'occupation de la mémoire physique disponible, la vitesse d'exécution et la consommation d'énergie. Notons que ces critères font référence à la performance du logiciel à l'exécution, d'autres critères comme le temps consommé par la compilation du logiciel ne sont pas ici pris en considération.

3.2.1 Occupation mémoire

L'occupation de la mémoire physique a toujours été un critère important d'optimisation dans l'ingénierie logicielle. En effet, elle a un impact sur le prix du stockage de données et du code, ainsi que sur les temps d'accès aux informations et donc les performances temporelles des applications. Cela est particulièrement vrai dans le cadre des système embarqués, où les ressources physiques et logicielles disponibles restent limitées.

Nous faisons la distinction entre deux familles de stratégies d'optimisation concernant ce critère : la première agit de façon statique et vise la réduction de la taille du code et/ou des données, notamment avant le déploiement des composants logiciels ; la deuxième se concentre sur la gestion dynamique de la mémoire disponible et l'optimisation de la consommation à un instant t de l'exécution du logiciel [Kandemir 2005]. Dans le cadre de cette thèse nous nous focaliserons sur la première des familles.

3.2.2 Temps d'exécution

La détermination du temps d'exécution des méthodes d'une interface, et plus généralement des chemins d'exécution du logiciel, est une étape nécessaire pour la validation des logiciels embarqués. L'optimisation (dans la plupart des cas, la minimisation) de cette mesure temporelle est un enjeu prépondérant dans le cadre des systèmes à fortes contraintes temporelles comme les dispositifs multimédia [Zervas 1998] et les systèmes à temps réel dit « dur ».

La méthode la plus commune d'estimation des limites de temps d'exécution consiste à mesurer le temps d'exécution d'un chemin et à calculer les temps minimal

et maximal. Ces mesures auront une tendance à surestimer les meilleures et pires temps d'exécution (le *Best Case Execution Time*, BCET et le *Worst Case Execution Time*, WCET), qui sont des mesures plus adaptées aux besoins du temps réel [Wilhelm 2008].

3.2.3 Consommation d'énergie

L'efficacité de consommation énergétique et l'allongement de l'autonomie des systèmes embarqués nomades sont devenus récemment peu des sujets d'étude importants pour le monde industriel et académique. La performance des batteries, sur lesquels s'appuie l'autonomie des dispositifs embarqués, ne s'est multipliée que d'un facteur de 4 en 20 ans [CAS 2009], soit peu en rapport à l'augmentation de la puissance de calcul. A cela, il faut ajouter des motivations d'ordre écologique, puisque la consommation d'énergie dans le monde due aux appareils électroniques devient de plus en plus importante [1].

L'identification des sources principales de consommation, nécessaire pour concevoir des stratégies d'optimisation, est une tâche difficile dans le contexte des systèmes embarqués, compte tenu de leur hétérogénéité par rapport aux fonctionnalités et aux architectures matérielles et logiciels déployées. Basés sur des expériences variés, nous pouvons identifier les taches de communication (notamment sans fil, e.g. radio), l'affichage graphique (e.g. écrans) et l'exécution du flux d'instructions lié aux applications, comme les principales sources de consommation [2]. De cela se dérivent des stratégies d'optimisation qui visent à contrôler le fonctionnement de tel ou tel composant.

On ne considère pas ici les politiques d'optimisation énergétique mais on retiendra le besoin de configuration et de gestion des composants de bas-niveau, typiquement au travers des pilotes de périphériques.

L'optimisation au sein des systèmes embarqués est un domaine vaste. Nous ne nous intéressons pas ici à l'optimisation du code applicatif mais à l'optimisation de la génération de composants, soit de la structure du système, à partir des descriptions de haut niveau. Cette optimisation est vue au niveau de l'occupation mémoire et des temps d'exécution.

3.3 Compromis

L'optimisation du logiciel se focalise le plus souvent sur un ou deux aspects de la performance, en fonction des usages des applications et des caractéristiques des

1. 20% de l'énergie à Amsterdam est utilisée par des systèmes de télécommunication, 13% de la consommation électrique aux États Unis provient de l'utilisation d'applications d'ordinateur, Internet inclus [Abril Garcia 2005].

2. Crk, Albinali, Gniafy et Hartman [Crk 2009] ont mesuré la consommation d'un téléphoné mobile qui supervise des activités physiques via des capteurs divers. La proportion de consommation d'énergie par composant est de 38% pour l'affichage graphique, 33% pour la communication Bluetooth, 19% pour l'application et 10% pour d'autres activités liées aux fonctionnement du téléphone.

plate-formes d'exécution. La multiplicité des critères conduit à des compromis dans lesquels un ensemble d'aspects sont optimisés au détriment d'autres.

Il est important de signaler que ces compromis concernent non seulement des critères liés à la performance à l'exécution, sujet principal de cette étude, mais aussi d'autres critères d'évaluation des logiciels. Nous avons identifié trois compromis qui nous semblent les plus pertinents dans le cadre de cette thèse :

- **Conflits entre critères de performance**. Fait référence aux compromis entre les critères présentés au chapitre 3.2 comme ceux étudiés dans [Kandemir 2005, Leupers 1999, Naik 2001, Badea 2008, Baynes 2001]. Un exemple illustratif est celui des fonctions *inline* [Leupers 1999] : cette technique, qui s'applique à la compilation, ajoute le code correspondant à une fonction `f` à l'intérieur d'une ou plusieurs fonctions afin de réduire le temps d'exécution de ces dernières, au détriment de l'occupation mémoire du logiciel.
- **Ressources consommés dans le processus d'optimisation**. La quantité de ressources utilisées pour mener les activités d'optimisation peut rendre prohibitif le processus d'optimisation. Cela est le cas pour les techniques d'optimisation à l'exécution (cf. 3.5.3) comme l'optimisation dynamique [Duesterwald 2005, Voss 2000] et la compilation *Just-In-Time* (JIT) [Chambers 1989, Badea 2007], dans le cadre des systèmes embarqués à ressources limitées.
- **Optimisation vs. Évolution du logiciel**. Les optimisations agressives sur les architectures à composants (e.g. fusion ou disparition de composants et d'interfaces [Balasubramanian 2008]) et sur les implémentations (e.g. transformation radicale du code source ou des automates comme dans [Callou 2008]) peut rendre impossible l'identification et l'analyse à l'exécution des sous-parties du système. Les fonctions *inline* [Leupers 1999] est à nouveau un bon exemple : une fonction `f` qui est inlinée « disparaît » à l'exécution, ce qui empêche toute tâche d'évolution (e.g. maintenance, correction de *bugs*) qui concerne cette fonction, car elle n'est plus détectable. Ces activités sont nécessaires au débogage et à l'évolution du logiciel.

Ces compromis sont pris en considération tout au long de ce chapitre, lorsque les techniques d'optimisation du logiciel sont évaluées.

3.4 La performance des composants logiciels et leur optimisation

Mesurer, prédire et/ou contrôler un ou plusieurs aspects de la performance d'un composant logiciel implique une analyse approfondie non seulement de son implémentation, mais aussi du contexte de compilation, de déploiement et d'exécution. Koziolok [Koziolek 2010] identifie 5 facteurs qui influencent la performance des composants logiciels :

i. **L'implémentation des composants**. Les fonctionnalités décrites par les interfaces offertes peuvent être implémentées de diverses manières, avec des différents profils de performance par rapport à un ou plusieurs critères.

ii. **Les services requis**. La performance d'un composant est liée aux performances des composants auxquels ses interfaces requises sont connectées.

iii. **Le profil d'utilisation**. La performance d'un composant varie en fonction de son utilisation, soient les valeurs des paramètres d'entrée des méthodes appelées (interfaces serveur).

iv. **Les plates-formes sous-jacentes**. La performance des composants est liée à celle des plates-formes logicielles et matérielles sous-jacentes (exécutifs et pilotes, intergiciels).

v. **Les conflits de ressources**. Les temps d'attente pour accéder à des ressources logicielles ou matérielles limitées affectent la performance du composant.

Dans le cadre de nos travaux de recherche, nous nous intéressons aux applications construites à partir de l'assemblage de composants. Les travaux de Koziolok se focalisant sur les composants primitifs, nous avons identifié deux facteurs additionnels qui affectent la performance des assemblages de composants logiciels, i.e. des composites :

vi. **Les sous-composants**. La performance d'un composite est liée à la performance individuelle de ses sous-composants. L'assemblage interne détermine la nature de cette relation.

vii. **L'implémentation de la « glue »**. La performance liée au code « glue » généré par les chaînes de compilation afin d'assurer l'interopérabilité entre les composants influence significativement la performance du système dans son ensemble [Becker 2006].

Une connaissance détaillée de ces 7 facteurs permet non seulement de mesurer ou prédire la performance d'un composant, mais aussi de déterminer l'impact que chacun a sur la performance de celui-ci, et donc de proposer des stratégies pour l'optimiser. Néanmoins, ces facteurs sont plus ou moins bien connus à différentes étapes du cycle de vie des composants, ce qui va déterminer dans une grande mesure la nature des techniques d'optimisation à mettre en œuvre à chacune de ces étapes.

Aux deux chapitres suivants, nous présentons des techniques d'optimisation qui concernent les critères décrits au chapitre 3.2 et qui agissent sur un ou plusieurs facteurs parmi ceux présentés dans ce chapitre. Le chapitre 3.5 décline ces techniques en fonction de la phase du cycle de vie où elles sont appliquées, alors que le chapitre 3.6 se focalise sur les techniques mises en œuvre par les plates-formes de développement à base de composants présentées au chapitre 2.2.4.2 de cette thèse.

Nous faisons référence aux facteurs identifiés dans ce chapitre par l'insertion des chiffres notés entre parenthèses ((i) à (vii)) qui correspondent à l'énumération donnée précédemment, lorsque la technique d'optimisation référencée fait usage de la connaissance du facteur correspondant.

3.5 Optimisation des performances en fonction du cycle de vie

Avant de s'intéresser à l'optimisation des modèles à composants, nous présentons un recueil non-exhaustif des techniques d'optimisation du logiciel, déclinées par rap-

port aux phases du cycle de vie (telles que définies au chapitre 2.2.3 et dans la figure 2.3) où elles sont appliquées. Notre sélection se centre sur les techniques utilisées dans des processus de développement des systèmes embarqués, ainsi que sur celles intégrées aux méthodologies de conception de haut niveau. Elle n'est pas néanmoins restreinte à ces domaines.

3.5.1 Conception

Au niveau des implémentations des composants (i) de nombreuses techniques ont été proposées. Leur efficacité dépend fortement de la quantité d'information sur les plates-formes logicielles et matérielles sous-jacentes (iv) dont elles disposent.

Certaines approches [Gerber 2002, Opt 2008, Hsieh 2004] reposent sur le savoir-faire des développeurs vis-à-vis leurs compétences en algorithmique et leur niveau de connaissance des chaînes de compilation spécifiques. D'autres se focalisent sur un sous-système matériel spécifique, comme dans [Panda 2001] pour le cas de la mémoire physique ou dans [Hong 2002] pour la mémoire des DSPs. Un bon nombre d'études [Becker 2008, Per 2002, Zhang 2009, Kwong 2010] se focalisent sur le choix des composants à inclure dans l'application. La performance des composants est calculée ou estimée à l'aide de modèles d'utilisation de ressources et/ou des couches logicielles et matérielles sous-jacentes [3].

En règle générale, la dépendance vis-à-vis les chaînes de compilation et les plates-formes sous-jacentes empêche l'extension des techniques d'optimisation sur un univers plus large de logiciels. Dans le cadre de cette thèse nous ne considérons pas cette famille de techniques d'optimisation.

Ces techniques se distinguent de celles appliquées à la mise en œuvre du fait qu'elles ne sont pas appliquées automatiquement mais fournies aux développeurs comme guides de programmation ou comme algorithmes d'optimisation du logiciel.

3.5.2 Mise en œuvre

A ce stade du développement une partie des caractéristiques des plates-formes logicielles et matérielles sur lesquels les composants s'exécutent (iv) sont connues (e.g. l'architecture du processeur qui exécutera le code binaire ou les services présents sur le site d'exécution). On peut ainsi optimiser agressivement la transformation de l'implémentation des composants (i), i.e. la génération de code assembleur.

Sur ce point, les approches existantes sont très variées : certains compilateurs comme GCC [FSF 2010b] offrent des *flags* de compilation prédéterminés qui concernent des modifications au niveau algorithmique et des optimisations qui dépendent de la famille de processeur sur lequel le programme sera exécute. D'autres chaînes de compilation sont dédiées à une plate-forme matérielle spécifique et intègrent des modèles de celle-ci qui permettent d'optimiser plus finement la performance,

3. Cette approche est utile non seulement à l'optimisation de la performance du logiciel, mais aussi à la gestion d'autres aspects économiques du développement logiciel. Elle est fondée sur le constat qu'une modification effectuée sur un système après son déploiement est bien plus coûteuse (en termes d'heures d'ingénieur de développement) qu'une modification au moment de la conception de celui-ci [Tassey 2002].

par exemple au niveau de la consommation d'énergie [Šimunić 2000, Lee 2001, Tiwari 1994] ou de l'occupation mémoire [Clausen 2000, Schultz 2003].

D'autres approches se focalisent sur le code binaire issu de la compilation. Elles ajoutent une passe d'optimisation sur des modules binaires de granularité arbitraire, mais le plus souvent la plus grosse possible (idéalement tout le programme) pour prendre en compte toutes les connections entre les modules (vi, vii). Dans [De Bus 2003, De Sutter 2005, Sutter 2007] une analyse du flux de contrôle du binaire exécutable mène à l'enlèvement de code « mort » et à l'élimination de code dupliqué. Ces techniques agissent directement sur le code assembleur dépendant de la plate-forme matérielle, ce qui les rend peu portables. A notre connaissance, elles n'ont pas été intégrées à des plates-formes de développement de composants logiciels.

Lors du déploiement, on connaît avec certitude les services disponibles sur la plate-forme et il est donc possible d'optimiser la connexion des interfaces clientes des composants (ii). Dans [Reussner 2003] on introduit la notion de contrats paramétrés. Le code « glue » (vii), correspondant aux implémentations de ces contrats, est modélisé par des machines d'états finis qui réagissent à l'environnement de déploiement du composant afin d'optimiser la fiabilité des assemblages. Cette technique n'a pas été implémentée dans le contexte des systèmes embarqués, nous estimons que le surcoût en occupation mémoire lié aux implémentations des contrats est non négligeable.

Dans [Balasubramanian 2008], l'assemblage des composants (vii) est optimisé par rapport à l'occupation mémoire, via la fusion des composants une fois que leur déploiement, généralement distribué en plusieurs noeuds d'exécution, est connu. Une bonne partie des méta-données que la chaîne de compilation génère pour chaque instance de composant est supprimée.

3.5.3 Exécution

Pour la plupart des systèmes, ce n'est qu'au moment de l'exécution du logiciel qu'on connaît en détail le profil d'utilisation des composants (iii). En particulier, l'implémentation d'un composant peut être optimisée par rapport à l'ensemble des paramètres d'entrée qui ont la plus haute probabilité d'occurrence. Pour ce faire, les systèmes doivent avoir la capacité d'adapter dynamiquement le code binaire en fonction des informations récoltées à l'exécution [Srikant 2002].

Des nombreuses techniques qui exploitent cette information ont été proposées. Parmi celles-ci on trouve l'optimisation dynamique [Duesterwald 2005, Voss 2000] qui transforme dynamiquement le code binaire, la spécialisation dynamique de programmes ou *évaluation partielle* [Jones 1996, Howell 1999] et la compilation JIT d'une représentation intermédiaire du code [Chambers 1989, Badea 2007]. Des compromis entre le critère qu'on cherche à optimiser et d'autres critères de performance sont souvent soulevés : par exemple, le temps consommé pour compiler à la volée une partie du logicielle et l'espace mémoire consommé par l'infrastructure de compilation dynamique peuvent s'avérer prohibitifs.

Conclusion. Nous constatons l'hétérogénéité des techniques d'optimisation exis-

tantes, ce qui est en concordance avec la grande variété d'applications et des plates-formes logicielles et matérielles que l'on trouve dans le contexte des systèmes embarqués. Nous remarquons également la manière dont les différentes techniques font usage des informations rendues disponibles au fur et au mesure du cycle de vie du logiciel.

Afin d'étudier plus en détail l'optimisation des composants logiciels, dans le paragraphe suivant, nous évaluons les optimisations effectuées par les quatre plates-formes de développement à base de composants présentées au chapitre 2.2.4.2 de cette thèse.

3.6 Optimisation dans les plates-formes de développement de référence

3.6.1 Préambule sur l'impact de la compilation de bas niveau

Comme cela a été évoqué au chapitre 2.2.4.3 de cette thèse, les plates-formes de développement de reférence font usage de chaînes de compilation classiques pour produire des binaires exécutables. Typiquement, du code C est généré par les compilateurs propres aux chaînes de développement à partir des description de haut niveau d'abstraction. Ce code C est ensuite compilé en binaire exécutable par des compilateurs classiques de bas niveau, tels que GCC ou IAR. La figure 3.1 illustre ce processus.

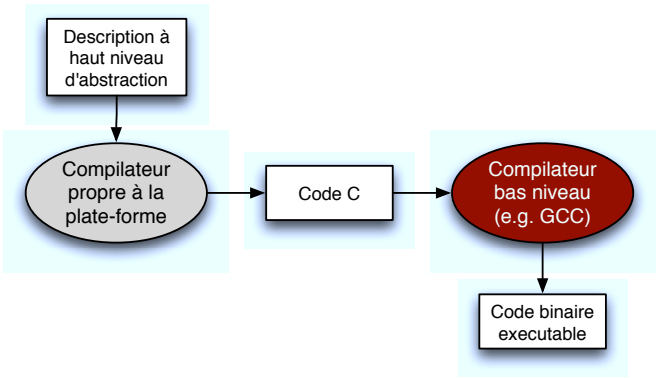


FIGURE 3.1 – Vue simplifiée du processus de compilation des plates-formes de développement de référence.

Les compilateurs de bas niveau intègrent des techniques d'optimisation variées qui pour la plupart concernent des implémentations binaires différentes d'un même comportement (i) et impactent la performance du logiciel par rapport aux critères définis précédemment : occupation mémoire, temps d'exécution et consommation énergétique. Les plates-formes de développement de référence utilisent le compilateur GCC. GCC définit 5 niveaux d'optimisation [FSF 2010b]. Chacun d'eux met l'accent sur un ou plusieurs critères de performance :

- *O0* : Réduction du temps de compilation (niveau par défaut).
- *O1* : Minimisation du temps d'exécution et de l'occupation mémoire. Seulement des optimisation qui n'impliquent pas un compromis entre ces deux critères sont appliquées. Les optimisations qui augmentent significativement le temps de compilation ne sont pas appliquées.
- *O2* : Comme O1 mais en appliquant aussi les optimisations qui augmentent le temps de compilation.
- *O3* : Comme O2 mais appliquant des optimisations plus agressives comme la modifications des boucles, la réutilisation des calculs et l'*inline* de fonctions.
- *Os* : Minimisation de l'occupation mémoire, même si le temps d'exécution de programmes est augmenté.

Une exécution plus fine des techniques d'optimisation est possible en activant des flags de compilation. GCC offre plus de 100 flags de compilation et au moins 50 concernant directement l'optimisation du logiciel [FSF 2010a]. La plupart d'entre eux sont indépendants tant des plates-formes matérielles sous-jacentes comme des langages de programmation, puisque les transformations se font sur le RTL (*Register Transfer Language*), un langage intermédiaire entre celui de haut niveau et l'assembleur du processeur. Par exemple, rien que sur l'*inline* de fonctions, GCC offre 8 flags qui s'appliquent directement au langage C : `-fno-inline` fait que le compilateur ignore le mot clé `inline`, `-finline-small-functions` demande au compilateur d'*inliner* les fonction « simples » (c'est au compilateur de décider les fonctions a qui on peut ajouter cet adjectif). Il existe aussi les flags suivants (non détaillés ici) : `-findirect-inlining`, `-finline-functions`, `-finline-functions-called-once`, `-fearly-inlining`, `-finline-limit=n`, `-fkeep-inline-functions` et `-fpartial-inlining`.

Les plates-formes de développement étudiées utilisent les optimisations offertes par les compilateurs classiques de deux façons. Certaines comme TinyOS ou Fractal/Think permettent à l'utilisateur de définir des niveaux et des flags d'optimisation pour le compilateur classique, dans ce sens elles agissent comme un intermédiaire entre le développeur et le compilateur de bas niveau, même si le premier n'a aucun contrôle sur le code qui est généré par les chaînes de compilation. D'autres comme Koala implémentent des techniques d'optimisation dont l'efficacité dépend des optimisations faites par les compilateurs de bas niveau. Tel est le cas des interfaces de diversité à valeur constante.

Nous nous focalisons maintenant sur les techniques d'optimisation propres aux plates-formes de développement de référence qui sont appliquées indépendamment des compilateurs de bas niveau utilisées.

3.6.2 TinyOS

Censée s'exécuter sur une plate-forme matérielle très contrainte, une application basée sur TinyOS est automatiquement optimisée de façon agressive. Les techniques d'optimisation considérées visent principalement la réduction de l'occupation mémoire des applications et sont menées lors de la compilation de celles-ci.

- **Enlèvement des fonctions non accessibles** : le compilateur nescc construit, basé sur l'assemblage des composants (vi), un graphe des connexions

qui est ensuite utilisé pour déterminer quelles fonctions ne sont pas invoquées dans le code source de l'application. Ces fonctions ne sont pas incluses dans le code généré, diminuant l'espace mémoire requis pour le déploiement de l'application.

- **Code monolithique** : le compilateur construit, à partir des implémentations de chacun des composants, un code d'implémentation monolithique (i) auquel il applique des techniques d'optimisation classiques comme la propagation de constantes, l'élimination de sous-expressions communes et l'enlèvement du code mort.

- **Optimisations cross-composants** : la construction d'un code monolithique permet aussi d'enlever le code source qui correspond à la « glue » (vii) représentant le croisement des frontières des composants. Les appels des fonctions sont ainsi optimisés avec, au maximum, un seul appel de procédure qui, dans une plate-forme matérielle ATMEL basée sur un processeur ARM7 consomme aux alentours de 8 cycles d'horloge [Corporation 1999, Corporation 2010]. Les optimisations cross-composants vont jusqu'à *inliner* des fonctions et des composants dans d'autres fonctions et composants.

Comme résultat de ces optimisations, le modèle à composant disparaît après la compilation du système : il s'avère extrêmement compliqué de suivre la trace d'un composant (défini à la phase de conception) dans le code binaire qui s'exécute. L'évolution du système se voit donc affectée par l'agressivité des optimisations appliquées. Puisqu'elles sont effectuées de façon systématique, le développeur n'a pas de choix possible concernant le dégré d'optimisation de son application.

3.6.3 CAmkES

Le compilateur CAmkES transforme les entités du modèle à composants en entités propres à Iguana/L4, la plate-forme logicielle sous-jacente. Par conséquent, la performance d'une application CAmkES est très proche à celle d'une application statique non-composant qui s'exécute sur cette plate-forme. CAmkES implémente une seule technique d'optimisation :

- **Disparition des composites** : les connexions vers les interfaces serveur des composites sont systématiquement dirigées vers les sous-composants adéquats. Étant donné que les composites ne contiennent aucune implémentation, cela équivaut à faire disparaître les composites dans le binaire issu de la compilation et donc à ne pas générer du code « glue » (vii) représentant les frontières entre composants.

Les développeurs de CAmkES partent du principe que la performance d'une application Iguana/L4 est satisfaisante, et sous cette hypothèse, ils estiment que le surcoût dû au modèle à composant est acceptable. Nous ne disposons pas d'une évaluation des performances de Iguana/L4 par rapport à d'autres systèmes d'exploitation pour l'embarqué.

3.6.4 Koala

Comme cela a été présenté au chapitre 2.2.4.2 page 27, le compilateur Koala effectue de manière systématique des optimisations concernant les interfaces de diversité et les switches.

- **Enlèvement des interfaces de diversité** : lorsque les propriétés dont l'interface de diversité fait référence ont des valeurs constantes, les appels aux fonctions sont remplacés par ces valeurs. Cela permet d'effectuer des optimisations en s'appuyant sur celles effectués traditionnellement par les compilateurs classiques. Par exemple, pour le code suivant :

  ```
  if (div_param()) {
          ...
  }
  ```

  Dans le cas où la propriété qui correspond à la fonction `div_param()` a une valeur constante 0, un compilateur comme GCC enlèvera automatiquement le bloc d'instructions qui serait exécuté dans le cas contraire.
- **Simplification des connecteurs du type *switch*** : lorsque l'interface de diversité qui contrôle le *switch* est évaluée constante à la compilation, celle-ci est simplifiée en une connexion directe entre le composant client et le composant serveur adéquat, et donc en un appel de fonction classique.

En plus de ces deux optimisations, le fait de marquer des types de composants comme multi-instanciables entraîne la modification des entêtes des fonctions des interfaces serveur afin d'introduire un paramètre qui lie le code de la fonction à une structure de données propres à chaque instance du type de composant. Cela évite la duplication de code et minimise l'espace mémoire consommé par l'application.

A exception de cette dernière, les optimisations dans Koala sont effectuées de manière systématique. Les techniques présentées se focalisent sur les implémentations du code « glue » généré à la compilation (vii).

3.6.5 Fractal/Think

La plate-forme Fractal/Think n'implémente aucune technique d'optimisation proprement dite. Néanmoins, deux actions qui conduisent indirectement à des optimisations en termes d'occupation mémoire sont effectuées pendant le processus de compilation :

- **Composants multi-instance** : à différence de Koala, tous les composants Think sont multi-instanciables. Cela évite la duplication de code dans le cas où plusieurs instances d'un type de composant sont incluses dans l'application. Néanmoins, dans le cas contraire la chaîne de compilation génère du code inutile.
- **Composants partagés** : le modèle Fractal définit des composants partagés, i.e. des composants qui peuvent être inclus dans plusieurs composites différentes. Cette notion est utile pour la modélisation de ressources communes à tout le système. Les connexions vers les interfaces serveur de ces composants sont simplifiées à des liaisons simples, ce qui peut être assimilable à un enlèvement des frontières entre les composants client et serveur (vii).

La nature très dynamique des applications Fractal/Think et l'absence des techniques d'optimisation dans la chaîne de développement provoquent des surcoûts considérables en termes d'occupation mémoire et de temps d'exécution. Une étude approfondie concernant le surcoût dû à la prise en compte des aspects extra-fonctionnels et notamment de la réconfiguration dynamique des assemblages de composants a été menée dans [Navas 2009b]. Les résultats obtenus montrent que l'inclusion de cet aspect peut provoquer une occupation mémoire double de celle d'un logiciel non reconfigurable [4].

3.7 Synthèse sur l'optimisation

Un certain nombre d'éléments ressortent de l'étude des techniques d'optimisation existantes. D'abord, le nombre important des techniques retrouvées dans la littérature est un indicateur de la pertinence de l'optimisation du logiciel et de l'importance de l'amélioration des performances à l'exécution, malgré les améliorations continuelles des performances au niveau matériel.

Nous constatons une relation directe entre l'étape du cycle de vie où les optimisations sont appliquées et les ressources consommées pour effectuer ces optimisations. Les techniques appliquées aux dernières étapes de la compilation, au déploiement et à l'exécution du système agissent majoritairement sur les composants représentés en code binaire exécutable. Cette représentation n'est pas facile à manipuler et soulève le problème du compromis entre le critère de performance recherché et les efforts nécessaires pour obtenir les résultats souhaités. De plus, ces techniques d'optimisation sont fortement liées aux plates-formes matérielles d'exécution, ce que les rendent peu portables [5]. De façon analogue, certaines approches qui agissent à la conception sur les représentations de plus haut niveau du logiciel (e.g. par le choix des composants en fonction de leurs performances estimées) sont à première vue plus simples à mettre en œuvre, mais demandent une connaissance approfondie des plates-formes matérielles et logicielles sous-jacentes, ce qui les rendent peu portables.

Dans le cadre de cette thèse nous nous focalisons sur les techniques d'optimisation appliquées à la mise en œuvre des composants logiciels. Nous remarquons, à partir de l'étude menée au chapitre 3.6, que les optimisations considérées par les plates-formes de développement ici référencées, sont effectuées à l'étape de compilation du système. Malgré l'hétérogénéité au niveau de la mise en œuvre des composants, nous distinguons un ensemble de propriétés communes concernant l'optimisation :

- **Optimisation de la structure du logiciel**. De manière générale, les plates-formes étudiées se focalisent sur l'optimisation de l'assemblage des composants, i.e. sur la structure du logiciel, et non sur le comportement des composants. Cela est dû à l'intention des plates-formes de développement de rester relativement génériques par rapport aux plates-formes matérielles et logi-

4. Cette propriété ne dépend pas de la chaîne de compilation utilisée. Dans [Porter 2010] des résultats similaires ont été obtenus en comparant le code binaire generé, pour un même système, par OpenCOM (application dynamique) et TinyOS (application statique).

5. Une alternative à explorer est celle de manipuler des codes d'instructions du type « assembleur » plus riches et communes à des diverses plate-formes, comme LLVA [Adve 2003] qui est utilisé dans l'exécution des LLVM [Lattner 2004] (*Low Level Virtual Machine*).

cielles, ainsi qu'aux outils de développement logiciel, notamment les langages de programmation et les compilateurs de bas niveau. Seul TinyOS effectue des optimisations qui demandent l'analyse du comportement des composants ; le prix à payer n'est pas négligeable : elle est la seule plate-forme qui fixe un langage autre que le C (le standard industriel) pour décrire le comportement des composants. De plus, nesC impose un ensemble de limitations aux programmeurs. Cela affecte la portabilité de la plate-forme ainsi que la réutilisation de code *legacy*.

- **Fixation précoce de la structure du logiciel**. Dans ce cadre, les plates-formes étudiées assument pour la plupart que l'assemblage de composants (et de manière générale, le logiciel) ne sera pas modifié après la compilation, et elles appliquent ainsi des techniques d'optimisation qui peuvent être catégorisées en :
  - **Optimisation des connexions**. L'attention est portée principalement sur l'optimisation des connexions entre composants et notamment sur l'enlèvement des frontières entre les composants : l'enlèvement des composites dans CAmkES et l'optimisation du code monolithique dans TinyOS. Cette technique d'optimisation peut même avoir comme résultat final la disparition de la connexion (e.g. l'*inlining* des fonctions).
  - **Enlèvement du code non accessible**. Les services des composants qui ne sont pas requis par aucun composant, ne sont pas ajoutées dans le binaire exécutable. Lorsque tous les services offerts par un composant correspondent à ce cas, le composant disparaît lui aussi. Seule TinyOS mène de façon explicite cette optimisation. Néanmoins, ce type d'optimisation peut être également menée par les chaînes de compilation de bas niveau comme GCC (cf. 6.3.3).
  - **Propriétés constantes**. L'optimisation des interfaces de diversité dans Koala illustre comment la fixation d'une propriété d'un composant entraîne des optimisations. TinyOS, via l'analyse du code monolithique et la propagation de constante qui est faite à cette étape, utilise aussi cette information pour optimiser le binaire exécutable.
  - **Composants mono/multi instanciables**. Fractal/Think et Koala prennent en compte la possibilité de partager le code entre composants du même type, en suivant deux approches opposées : alors que pour le premier les composants sont multi-instaciables par défaut, pour le deuxième ils sont mono-instances par défaut, et multi-instanciables si défini par les concepteurs. En fonction des caractéristiques de l'application le deux approches peuvent avoir comme résultat des optimisations sur le binaire exécutable.
- **Code « glue »**. Au niveau de leur mise en œuvre, les optimisations étudiées précédemment se font en exerçant un contrôle sur la génération du code « glue », qui assure l'interopérabilité entre les composants.
- **Compromis entre critères d'optimisation**. Les techniques ici étudiées illustrent les types de compromis définis au chapitre 3.3 :

- **Conflits entre critères de performance**. Outre le cas des fonctions *inline*, qui réduit le temps d'exécution au détriment de l'occupation mémoire, et qui est effectué par TinyOS, les composants multi/mono instanciables illustrent eux aussi des compromis entre critères de performance à deux niveaux : i) le partage de code entre composants de même type réduit l'occupation mémoire due à ceux-ci, mais peut augmenter le temps d'exécution de l'application, dans la mesure où il introduit un nouveau paramètre d'entrée aux méthodes partagées et donc augmente le temps d'invocation de ces méthodes ; ii) le gain en termes d'occupation mémoire dû au partage de code entre composants du même type peut se voir dépassé par l'occupation mémoire supplémentaire liée à ces composants compilés comme multi-instanciables. Koala introduit la possibilité de définir à la conception quels composants seront multi-instanciables, mais cela demande également des transformations du contenu des composants clientes, ce qui rend l'approche peu fidèle à la philosophie des composants logiciels.
- **Ressources consommées dans le processus d'optimisation**. Le fait de nous focaliser sur les techniques d'optimisation appliquées à la mise en œuvre des composants et non à leur exécution, évite des conflits de cet ordre.
- **Optimisation vs. Évolution du logiciel**. Puisque les techniques identifiées sont basées sur l'hypothèse du caractère statique de la structure du logiciel, le conflit entre l'optimisation et la capacité du système à évoluer après son déploiement est très présent. On distingue un premier groupe de plates-formes où des optimisations agressives sont menées à la compilation (TinyOS et Koala) et un seconde où le caractère dynamique des composants et pris en compte et où le nombre d'optimisations menées est bien plus limité (Fractal/Think et CAmkES).

- **Optimisations systématiques**. Enfin, nous constatons que les optimisations sont appliquées pour la plupart de manière systématique. Cela répond à une préoccupation par rapport aux limitations de ressources propre au domaine d'étude. Puisque l'optimisation agressive engendre des compromis vis-à-vis d'autres préoccupations extra-fonctionnelles comme l'évolution logicielle, nous considérons que le manque de contrôle par rapport aux optimisations appliquées constitue un inconvénient majeur, notamment pour l'évolution des logiciels après leur déploiement.

Ces considérations serviront de directives pour les contributions présentées dans la partie II de cette thèse. Notamment, et comme l'on verra dans le chapitre suivant, il y a une relation profonde entre la préservation du modèle à composants à l'exécution et les capacités d'évolution d'un système. Par conséquent nous prévoyons à ce stade que le compromis entre optimisation et évolution logicielle est particulièrement critique en ce qui concerne les applications basées sur des composants logiciels.

Chapitre 4

Evolution

Sommaire

4.1 Définition **49**
4.2 Activités propres à l'évolution **50**
4.3 Infrastructures d'évolution **51**
4.4 Évaluation des approches d'évolution existantes **52**
4.4.1 Considération de l'évolution au niveau des composants 53
4.4.2 Considération de l'évolution en dehors des composants 57
4.5 Synthèse sur l'évolution **61**
4.6 Synthèse de la première partie de la thèse **64**

Les systèmes embarqués sont étroitement liés à leur environnement, dont les processus physiques. La nature souvent imprévisible du monde réel dans lequel ces systèmes sont immergés, qui se manifeste par des phénomènes comme la modification des environnements d'opération ou l'apparition des nouveaux besoins fonctionnels, crée des besoins comme :

- Le déploiement de nouveaux services ;
- Le perfectionnement des services existants ;
- La mise à jour des composants déjà déployés sur les plates-formes matérielles ;
- La correction de *bugs* qui ont échappé aux vérifications menées lors des étapes précédentes à l'exécution dans le cycle de vie du logiciel ;
- La prise en compte et l'adaptation aux nouvelles fonctionnalités.

L'existence de tels besoins révèle le fait que le niveau de satisfaction assuré par le logiciel, par rapport aux spécifications fonctionnelles et extra-fonctionnelles définies à la conception, diminue progressivement. Le changement est alors une propriété inhérente à ce type de systèmes : une satisfaction continue des spécifications demande des changements en continu du logiciel.

De plus, les évolutions du système induites par ces besoins doivent suivre une exigence supplémentaire forte au niveau du déploiement :

- La diminution du *downtime*, i.e. le temps requis pour l'arrêt et la mise à jour du système ou d'une partie de celui-ci.

4.1 Définition

L'évolution du logiciel peut être définie comme le changement progressif et continu des propriétés ou caractéristiques d'une entité logicielle [1]. Dans notre cas

1. Les termes *évolution* et *maintenance* sont souvent utilisés indistinctement dans le contexte du logiciel, alors qu'ils existent un nombre important de différences sémantiques. Le terme *maintenance*

cette entité est un composant.

Il n'existe pas une vision unique de ce que le terme évolution représente dans le contexte des systèmes informatiques [Bennett 2000]. Le concept a été decliné par rapport à de dimensions telles que :

- Le **quoi**, i.e. la nature du phénomène d'évolution, ses causes et ses conséquences, étudié dans [Lehman 2001, Kemerer 1999].
- Le **pourquoi**, i.e. l'objectif poursuivi par l'évolution. Chapin et. al. [C 2001] identifient 12 types différents d'évolution en fonction des effets obtenus sur le logiciel après les activités d'évolution. Ces types sont regroupés en 4 groupes : *métier*, *propriétés du logiciel*, *documentation* et *interface de support*. Par exemple, une évolution du type correctif appartient au groupe *métier*, puisqu'elle a un impact direct sur l'expérience de l'utilisateur.
- Des dimensions plus techniques comme le **comment** et le **quand** de l'évolution. Buckley et. al. [Buckley 2005] proposent une taxonomie des caractéristiques des outils logiciels utilisés pour mener l'évolution, ainsi que des facteurs qui influencent ces outils.

Dans le cadre de cette thèse nous nous focalisons sur cette dernière dimension et notamment sur les techniques adoptées pour mener les activités propres à l'évolution des composants logiciels embarqués, i.e. le **comment** de l'évolution. Le **quand** est ici considéré à l'exécution du logiciel, i.e., l'objet d'intervention est le composant logiciel après son déploiement.

4.2 Activités propres à l'évolution

Les approches existantes implémentent des techniques d'évolution très variées. Nous identifions dans un premier temps trois activités nécessaires à l'évolution :

- **Observation**. L'observation de l'*état* du système permet d'identifier et surveiller les éventuelles causes d'une évolution. Nous définissons l'état comme un vecteur de propriétés mesurables à un instant donné qui caractérisent la situation d'un système [Polakovic 2008].
- **Raisonnement**. A partir de l'examen de l'état du système et de l'ensemble d'exigences qui le concernent et/ou des indicateurs de performance qu'en peuvent être dérivées, une analyse conduit à la planification des actions d'évolution qui satisferont ces exigences.
- **Intervention**. L'exécution des actions nécessaires pour faire évoluer le système. Il s'agit du changement des valeurs des paramètres qui affectent d'une façon déterminée le comportement du système.

La figure 4.1 montre les activités et l'ordre d'exécution de celles-ci : l'état du système observé participe au raisonnement sur les activités d'évolution (1). À partir de ce raisonnement une consigne d'intervention est déduite (2). L'intervention sur

est associé à la conservation du design du système et au remplacement ou traitement des sous-composants affectés ; dans la pratique, le logiciel ne se « détériore » pas et la maintenance logicielle consiste, le plus souvent, à modifier le design original. Le terme *évolution* est plus adapté dans notre cas, puisque il reflète la nature innovatrice et non seulement préservatrice du changement.

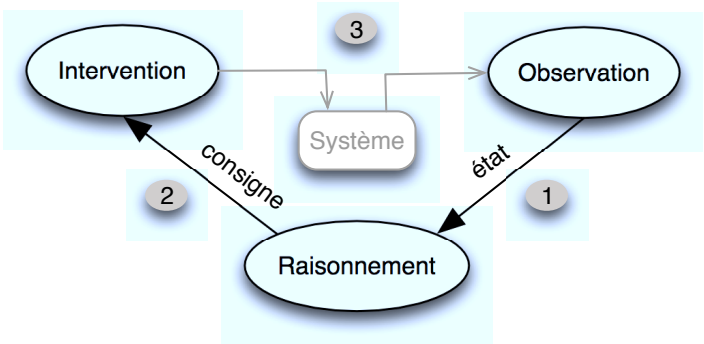


FIGURE 4.1 – Activités d'évolution et ordre d'exécution

le système modifie le comportement de celui-ci (3), le nouvel état résultant est observé et la boucle se referme. Nous constatons ainsi que l'évolution logicielle est un processus de contrôle qui a des nombreuses similitudes avec le contrôle en boucle fermée (*Feedback control*) issu de la théorie de régulation [Hellerstein 2004].

La maîtrise de l'évolution implique que l'observation, l'intervention mais surtout le raisonnement, s'appuient sur des représentations de haut niveau d'abstraction du logiciel qui évolue [Oreizy 1998, Oreizy 2008]. Au chapitre 2.2.4 nous avons constaté que le logiciel est représenté de différentes façons suivant l'étape du cycle de vie dans laquelle il se situe. Parmi elles, la représentation à la conception est celle de plus haut niveau d'abstraction et est, de ce fait, une bonne candidate pour la gestion des activités d'évolution.

Dans cette étude nous nous intéressons tout particulièrement à l'évolution à l'exécution. Dans ce cadre, l'évolution d'une représentation d'un composant à la conception entraîne des changements sur la représentation utilisé au moment de l'exécution du composant, changement qu'il faudra maîtriser.

Pour conclure, notre étude s'intéresse à l'intervention sur le système en exécution et à la manière dont les changements sur les représentations de haut niveau d'abstraction se répercutent sur celles du bas niveau.

4.3 Infrastructures d'évolution

Pour que les activités propres à l'évolution puissent être exécutées, il est pertinent de définir explicitement la nature des données manipulées dans la boucle de contrôle, ainsi que la façon dont les entités qui exécutent ces activités sont implémentées. Dans le cadre de notre étude nous nous focalisons sur :

- L'ensemble de propriétés qui composent l'état du système, ainsi que les mécanismes implantés pour extraire ces données du système en exécution et les intégrer dans la boucle de contrôle.
- L'ensemble des *points d'intervention*, i.e. l'ensemble des paramètres qui affectent d'une façon déterminée le comportement du système en exécution, ainsi que les effets sur le comportement du système provoqués par une action

effectuée sur ces points.
– Le comportement du contrôleur, i.e. la manière dont le raisonnement sur l'évolution est implémenté et intégré au système en exécution.

Le logiciel qui décrit ces activités constitue ce que nous appelons une **Infrastructure d'Évolution** ou **IdE**. Comme pour tout logiciel, on peut définir des critères qui caractérisent la performance des IdE. Nous en avons identifié quatre :

- La **richesse d'évolution** fournie par l'IdE, i.e. le nombre d'alternatives d'évolution du système que sont disponibles après l'intégration de l'IdE à celui-ci. Une *alternative d'évolution* est un état auquel le système pourrait arriver à partir d'un état donné, via une ou un ensemble d'actions d'évolution. Un système riche en alternatives d'évolution est mieux adapté à des environnements très variables.
- L'**impact sur la performance du système** que l'implantation de l'IdE provoque, impact qui est défini selon les critères du chapitre 3.2. Par exemple, le temps consommé par l'exécution des actions d'évolution peut avoir une conséquence non négligeable sur la performance du système si celui-ci est obligé de suspendre son exécution normale. De même, l'occupation mémoire est un critère à intégrer pour les systèmes embarqués.
- Le degré de **dépendance** et d'**entrelacement** entre les préoccupations liées à l'évolution et à l'implémentation de l'IdE, et les aspects purement fonctionnels du système. Idéalement l'évolution devrait être traitée comme un aspect extra-fonctionnel du développement, en modifiant au minimum les processus de développement habituels ainsi que l'exécution du logiciel.
- La **sûreté de l'évolution**, i.e. que l'IdE assure que le système aura un comportement cohérent et stable pendant l'exécution des actions d'évolution et après les changements effectués.

Comme tout logiciel, une IdE possède son propre cycle de vie composé des phases décrites au chapitre 2.2.3. Puisque l'IdE est destinée à être intégrée au système cible, la façon dont le cycle de développement de l'IdE se superpose sur le cycle de développement du logiciel embarqué est un facteur qui influence la performance de l'IdE.

Au chapitre suivant nous évaluons un ensemble de techniques d'évolution du logiciel qui nous semblent représentatives du domaine de l'embarqué. Nous utilisons les points ci-avant définis comme critères de différenciation de celles-ci.

4.4 Évaluation des approches d'évolution existantes

Nous classifions les approches d'évolution en deux familles qui sont examinées aux chapitres suivants. La première famille regroupe les approches qui traitent de l'évolution au niveau des composants, et dont les implémentations de l'IdE est étroitement liée au logiciel. Puisque les composants sont des entités utilisées à la conception, une conséquence est que les problématiques propres à l'évolution sont considérées à des stades précoces du cycle de développement.

La deuxième famille regroupe des approches qui pour la plupart font abstraction des architectures logicielles et dont l'IdE est implémentée de façons relativement indépendantes des composants déployés. Cela leur permet de considérer l'évolution à des stades tardifs du développement logiciel.

Nous privilégions les approches basées sur des composants logiciels natifs où le comportement des composants logiciels est décrit dans un langage de programmation qui est compilé et exécuté par un processeur, comme le langage C. Afin d'élargir l'étude, nous incluons dans notre recueil des techniques d'une nature distincte et notamment des techniques qui ne reposent pas sur l'utilisation d'un modèle à base de composants.

Par contre, nous excluons les travaux qui reposent sur l'exécution d'une machine virtuelle, qui sont certes nombreux, particulièrement dans le domaine des réseaux de capteurs [Brouwers 2009, Müller 2007, Michiels 2006, Xie 2006, Koshy 2005, Levis 2002]. Dans ceux-ci, des operations complexes composées de plusieurs instructions du processeur sont abstraites en *bytecodes* uniques qui sont ensuite interprétés. Concernant l'évolution, cette approche diminue considérablement la taille des paquets à transmettre (et donc l'énergie consommée dans la transmission) et le *downtime*. L'idée est intéressante, néanmoins, cette démarche n'est pas extensible à cause de i) la trop forte relation entre les *bytecodes* définies par la machine virtuelle et les besoins fonctionnels de l'application, évalués à sa conception, et ii) le très fort impact en termes d'occupation mémoire et de temps d'exécution que l'implantation d'une machine virtuelle implique. Nous soulignons néanmoins que des efforts sont en cours pour limiter ces inconvénients [Koshy 2008, Koshy 2009].

4.4.1 Considération de l'évolution au niveau des composants

Nous classifions les approches appartenant à cette famille en trois sous-familles :
- celles qui définissent à la conception des *modes d'opération* du système (cf. 4.4.1.1) ;
- celles où les politiques d'évolution sont encapsulées dans un composant dédié ;
- celles où les actions d'évolution et non le raisonnement sont définies à la conception.

4.4.1.1 Modes d'opération

AADL [Feiler 2004] (de l'anglais *Architecture Analysis and Design Language*) est un langage de description d'architectures standardisé par la SAE (de l'anglais *Society of Automotive Engineers*) et utilisé pour l'analyse et la conception d'architectures logicielles dans le cadre des systèmes embarqués critiques. Les travaux de Borde et. al. [Borde 2009] permettent la définition de **modes d'opération** de systèmes au moment de la conception de ceux-ci. Les actions d'évolution ont comme objectif la transition d'un mode d'opération à un autre, la consigne de chaque évolution étant le mode objectif.

Le raisonnement, dans ce cas l'évaluation de l'ensemble de conditions à remplir pour effectuer un passage d'un mode à un autre, est exhaustivement défini à la conception et son implémentation est générée par la chaîne de compilation. Un

composant nommé « automata » est associé à chaque instance de composant plausible d'évoluer. Celui-ci est chargé des actions d'évolution concernant l'instance de composant. Ces actions d'évolution consistent à exécuter les méthodes des interfaces orientées évolution qui sont offertes par les composants et implémentées dans leur « enveloppe ». Ces interfaces permettent la modification de leurs attributs et la redirection des connecteurs.

Richesse d'évolution. Les perturbations sur le système et les consignes d'évolution sont définies et modélisées à la conception du système. La richesse d'évolution est limitée aux modes d'opération définis à priori. Le système n'est pas adapté à des changements imprévisibles lors de son utilisation, ce qui s'explique par le domaine d'application visé (les systèmes critiques).

Impact sur la performance. La définition des modes d'opération permet la mise en place de techniques d'optimisation sur les opérations de transition de mode, ainsi que la minimisation de l'espace mémoire assignée aux composants. Néanmoins, puisque chaque composant est étiqueté par le mode d'opération pour lequel il est valide, le phénomène de duplication de code peut être rencontré. De plus, un surcoût est lié au fait d'embarquer un code mais aussi, à priori, toutes ses évolutions possibles.

Tissage d'aspects. L'encapsulation de l'aspect « évolution » dans les composants automates et au sein des enveloppes des composants, générés de façon dynamique à la compilation, assure un tissage minimal et une séparation des préoccupations fonctionnelles de celles liées à l'évolution, ce qui garantit la réutilisation des composants.

Sûreté. La définition de modes d'opération à la conception permet de générer le code de transition des modes qui assure un comportement stable pendant l'évolution du système, ainsi qu'un temps d'arrêt du système minimal. La cohérence de chacun des modes d'opération est vérifiée formellement avant l'étape de compilation du système.

4.4.1.2 Intergiciels

Un certain nombre d'approches proposent l'implantation d'une couche logicielle dédiée à l'aspect évolution. Cette couche observe l'état du système, détecte les conditions nécessaires ou suffisantes aux changements et mène les actions d'évolution correspondantes. Matthys et. al. [Matthys 2009] s'appuient sur le modèle à composants LooCI [Hughes 2009] et proposent un intergiciel qui analyse en continu les événements soulevés par l'exécution des composants. Le raisonnement se base sur des politiques préalablement chargées. Les actions d'évolution comprennent le blocage d'événements, le choix d'appel à des méthodes au *runtime* et le déclenchement d'événements qui seront ensuite traités par des composants dans l'architecture.

ESCAPE [Russello 2008] est un intergiciel similaire basé sur la plate-forme TinyOS. Grace et. al. [Grace 2006] proposent aussi une approche semblable basée

sur des politiques d'évolution définies dès la conception.

Richesse d'évolution. En principe les alternatives d'évolution sont celles qui sont décrites par la politique installée, donc relativement restreintes. Néanmoins, certaines IdE permettent d'installer de nouvelles politiques d'évolution à l'exécution.

Impact sur la performance. L'analyse en permanence de l'état du système a un impact important sur le temps d'exécution du logiciel. Selon les politiques, il se peut qu'à chaque déclenchement d'un événement, l'intergiciel doive raisonner sur la pertinence ou non d'intervenir le système, ce qui peut s'avérer très coûteux en fonction de la complexité de ce raisonnement. De plus, l'intergiciel a un impact non négligeable en termes d'occupation mémoire.

Tissage d'aspects. L'exécution du logiciel dépend non seulement des composants fonctionnels mais aussi des politiques opérées par l'intergiciel. Ainsi, après l'exécution d'une action d'évolution, le changement n'est pas traçable, puisqu'il se manifeste par un changement de la politique et non de l'architecture du logiciel. Également, ces approches imposent souvent des paradigmes d'exécution spécifiques – basé sur des événements dans le cas concret de [Matthys 2009].

Sûreté. Ce critère n'est pas considéré dans les travaux référencés. La politique de changement est censée prendre en compte cet aspect.

4.4.1.3 Points fixes d'intervention

Une autre approche possible est de définir des points fixes d'intervention dès la conception du système, ce qui correspond à définir l'ensemble des commandes qui peuvent être exécutées par les actions d'évolution et les entités qui effectueront le raisonnement.

Les *switches* du modèle Koala (cf. 2.2.4.2 page 27) illustrent cette solution. C'est à l'intérieur du *switch*, via l'interface de diversité, que l'action d'évolution est menée. Pourtant, c'est à l'extérieur de celui-ci, dans le composant qui implémente l'interface de contrôle, que la pertinence de l'évolution sera évaluée. La solution technique proposée par Koala est toutefois très primitive, elle n'offre pas une grande richesse d'évolution et exige que les conditions pour déclencher les actions d'évolution soient évaluées en permanence.

Concernant le modèle CAmkES (cf. 2.2.4.2 page 25), dans [Kuz 2007a] le « runtime », i.e. les composants qui sont déployés par défaut pour tout application basée sur ce modèle, et qui font partie de la plate-forme logicielle sous-jacente, est étendue afin de permettre la création dynamique de nouvelles instances de composants et de connecteurs. Leur approche est néanmoins fortement liée à des scénarios d'évolution que les auteurs considèrent génériques, mais qui leur permet de faire des hypothèses assez fortes sur l'IdE, afin de minimiser son impact en terme de ressources consommées. Notamment, les actions de découverte des nouveaux composants et d'introspection sur les composants existants ne sont pas prises en compte. Au final, pour utiliser les extensions proposées au modèle, le raisonnement sur le changement

doit être défini à la conception, ce qui limite la richesse des alternatives d'évolution.

En ce qui concerne Fractal/Think (cf. 2.2.4.2 page 29), les travaux de Polakovic et. al. [Polakovic 2006, Polakovic 2007] se focalisent sur la sûreté des reconfigurations des architectures logicielles à base de composants Fractal. Les auteurs utilisent les interfaces d'introspection et d'intercession des composants pour observer l'état et effectuer des changements au niveau des valeurs des attributs et des connexions entre composants. Le raisonnement sur l'évolution est distribué : des sous-composants dédiés agissent comme contrôleurs des réconfigurations dans chaque composant qui peut potentiellement évoluer, afin de vérifier les conditions qui déterminent la sûreté des actions d'évolution (e.g., aucun *thread* s'exécute dans un composant au moment du changement de celui-ci).

Les décisions d'évolution sont prises soit par les composants en exécution (et donc définies lors de la conception du système, comme pour CAmkES), soit par un agent embarqué qui déclenche les actions nécessaires, après avoir analysé l'architecture des composants en exécution. Cette deuxième démarche nécessite néanmoins une connaissance minimale du déploiement du système sur la plate-forme matérielle.

Richesse d'évolution. Puisque tous les composants fournissent des interfaces réflexives qui permettent des changements dynamiques des connexions et des valeurs des attributs, ainsi que le téléchargement de nouveaux composants, les alternatives d'évolution sont seulement limitées par la mémoire disponible dans le dispositif.

Impact sur la performance. Le surcoût en occupation mémoire produit par l'inclusion des interfaces réflexives devient très vite prohibitif, au fur et au mesure que le nombre de composants augmente et surtout lorsque leur granularité diminue. L'alternative est de limiter le nombre de composants qui fournissent des services d'introspection et intercession, au détriment de la richesse d'évolution. Cette alternative a été évaluée dans [Navas 2009b].

Tissage d'aspects. Dans la mesure où le raisonnement sur les actions d'évolution est entrelacé avec le code fonctionnel du composant, la réutilisation des composants est sérieusement affectée. On peut éviter cet écueil via de bonnes pratiques de conception (e.g., des implémentations dédiées à chacune des interfaces).

Sûreté. Dans [Leger 2009] l'auteur propose une définition de la cohérence des configurations dans le modèle Fractal, qui est utilisée pour garantir la fiabilité des actions d'évolution. D'un coté plus pratique, deux techniques pour assurer la sûreté des actions d'évolution sont introduites dans [Polakovic 2008] : le comptage dynamique de *threads*, inspiré des travaux autour MMLite [Helander 1998], où un composant peut évoluer lors qu'aucun *thread* s'exécute à l'intérieur de son implémentation ; et les intercepteurs dynamiques, inspirés de ceux mis en place dans le noyau K42 [Baumann 2005], qui reportent l'exécution des actions d'évolution au moment ou le composant se trouve dans un état « stable », i.e. un état sur lequel il peut évoluer de façon sûre. Ces deux techniques sont implémentées par des composants dédiés qui forment une enveloppe autour du composant qui peut potentiellement évoluer, et qui sont inclus lors de la compilation.

4.4.2 Considération de l'évolution en dehors des composants

Les approches appartenant à cette deuxième famille se focalisent sur la manière dont le système, déjà déployé, évolue d'un état à un autre. L'observation de l'état du système et l'activité de raisonnement sont moins étudiés par rapport aux approches de la première famille. En particulier, le raisonnement est supposé être mené par les administrateurs du système, i.e. des agents externes à celui-ci.

Dans ce contexte, la solution triviale à l'évolution du système en exécution est celle de remplacer entièrement le code binaire déployé par une nouvelle version de celui-ci qui incorpore les changements désirés. C'est le cas pour Deluge [Hui 2004], une plate-forme d'évolution basée sur TinyOS (cf. 2.2.4.2 page 23). Les inconvénients qui dérivent d'une telle approche au niveau de l'impact sur la performance du logiciel, notamment par rapport à la consommation d'énergie liée à la transmission de données et au temps d'arrêt du système, motivent l'exploration d'autres stratégies.

Nous classifions les approches appartenant à cette famille en trois sous-familles :

- celles où le logiciel est découpé en modules « gros grain » qui peuvent être remplacés à l'exécution ;
- celles où la même technique est appliquée mais une partie des actions d'évolution est menée en dehors du dispositif embarqué ;
- celles ou le code binaire est modifié à des granularités arbitraires.

Une caractéristique commune aux approches étudiées ici est qu'une partie des actions d'évolution sont exécutées durant l'exécution du système, à l'extérieur de celui-ci. Cela fait que l'IdE se trouve répartie en au moins deux sites, au minimum i) dans le dispositif embarqué (*in-site*), et ii) dans une machine distante liée au dispositif (*off-site*).

Puisque la considération des problématiques liées à l'évolution est faite après la conception du système, les développeurs sont libérés de la conception de l'IdE. Celles-ci sont, soit construites dynamiquement à la compilation, soit fixées à la conception, pour être ensuite intégrées au système lors des étapes de compilation ou du déploiement. L'IdE est ici intégrée de manière « tardive » au système.

4.4.2.1 Linkage dynamique

Contiki [Dunkels 2004] est un système d'exploitation pour les plate-formes matérielles très limitées en ressources, utilisé dans le cadre des réseaux de capteurs (WSN). Contiki supporte le déploiement et l'activation dynamique de modules [2][Dunkels 2006], qui sont transmis via le réseau en format ELF [ELF 1995], L'utilisation d'un tel format standard leur permet d'utiliser des outils existants et prouvés lors de la résolution des liens entre modules et le chargement de modules dans le dispositif embarqué.

L'état du système est défini par les modules installés et la connexion entre eux, même si Contiki ne dispose pas de modèles de plus haut niveau que le code source

2. Les modules, dans le cadre de Contiki, peuvent être associés au modules de chargement dynamique du noyau Linux [Henderson 2006]. Par rapport aux composants logiciels, ils ne fournissent aucune information sur les services offerts et requis, ni sur les propriétés qui déterminent leur comportement à l'exécution.

pour décrire les dépendances entre modules. Les actions d'évolution sont limitées au remplacement d'un module par un autre.

L'IdE est intégrée au système au déploiement : un module dédié au *linkage* dynamique des modules fonctionnels constitue la seule composante embarquée de l'IdE. Il se charge de la résolution des liens à l'aide d'une table de symboles générée à l'édition des liens du système.

D'autres plates-formes comme SOS [Han 2005] et FiGaRo [Mottola 2008] suivent une démarche similaire à celle de Contiki. Dynamic TinyOS [Munawar 2010] intègre la résolution dynamique de liens dans la plate-forme TinyOS. Dans ce dernier cas, les composants du système d'exploitation sont eux aussi remplaçables, ce qui n'est pas le cas pour Contiki.

Richesse d'évolution. Les alternatives d'évolution sont multiples, dans la mesure où chacun des modules peut être remplacé par une nouvelle version. Néanmoins, l'objet d'évolution est le module, qui a une granularité assez importante. En conséquence, si l'on veut modifier la valeur d'un attribut d'un module, on est obligé de télécharger une nouvelle version de celui-ci et de remplacer la version en exécution. Une autre conséquence de cette approche est que les alternatives d'évolution sont certes nombreuses, mais d'une certaine façon figées à la compilation par la définition des frontières des modules applicatifs.

Impact sur la performance. La relation entre richesse d'évolution et granularité des modules implique qu'un changement minime sur le comportement d'un module peut entraîner la transmission et le stockage d'une grande quantité des données et du code binaire, ce qui a des conséquences sur le consommation d'énergie et l'occupation mémoire. Une alternative est de définir des modules à granularité très fine. Ceci oblige à stocker un grand nombre de méta-données, notamment de tables des symboles, dans la mémoire. Concernant la consommation d'énergie durant la transmission de données, le format ELF n'a pas été conçu pour des architectures matérielle autres que 32 bits, ce qui produit des surcoûts de transmission inutiles dans la mesure où certains champs transmis sont prévus pour 32 bits mais n'occupent que 8 bits pour les plates-formes utilisées dans les WSN. Contiki propose une modification du format, le CELF (Compact-ELF) avec lequel ces surcoûts sont réduits.

Tissage d'aspects. Les aspects évolution sont encapsulés dans la table de symboles du système et un module dédié au *linkage* des nouveaux modules. Les aspects évolutifs et fonctionnels sont ainsi bien séparés.

Sûreté. Ce critère n'est pas considéré dans les travaux référencés. L'exécution du système doit être arrêtée lors de l'exécution des actions d'évolution.

4.4.2.2 Pre-linkage off-site

Afin de réduire l'impact du *linkage* dynamique sur la performance du système, notamment au niveau de la taille des données à transmettre et du temps de trai-

tement du code téléchargé, certaines approches utilisent des informations extraites lors de la phase de déploiement du système.

Shen et Chiang [Shen 2007] effectuent un pre-*linkage* des nouvelles versions des composants en utilisant la table de symboles du binaire en exécution, afin de résoudre les dépendances vis-à-vis les composants déjà déployés. Cette résolution partielle des liens est une étape supplémentaire du déploiement de nouveaux composants. Dans le dispositif embarqué, une IdE minimale constituée des composants dédiés offre une interface de gestion permettant l'installation et l'activation de ces composants. Le chargement de composants dans la mémoire est simplifié par l'utilisation d'un processeur intégrant une unité de gestion de mémoire, ce qui rend leur approche dépendante des architectures matérielles.

Guo et. al. [Guo 2008] suivent une démarche similaire et utilisent le *mapping* de la mémoire du système (au lieu de simplement la table des symboles) pour *pre-linker* les nouveaux composants et définir l'emplacement où ces composants seront chargés en mémoire ; leur approche est par conséquent moins dépendante des plates-formes matérielles. Guo suit la philosophie de Lehman [Lehman 1985], pour qui la conception du système et son évolution sont une seule activité de développement : dans leur approche la réconfiguration est traitée de la même façon que le processus de configuration initiale des composants.

Richesse d'évolution. Comme l'approche précédente, la richesse d'évolution est inversement proportionnelle à la granularité des composants, qui est figée à la conception du système. Dans [Guo 2008] la granularité des composants peut être assez fine, allant jusqu'au *Function Block* [John 2010] qui comporte une unique méthode et les données associées à celle-ci.

Impact sur la performance. Par rapport au *linkage* dynamique *in-site*, le processeur embarqué est libéré de la tache de résolution de dépendances des nouvelles versions des composants, ce qui a un impact favorable sur le temps d'exécution (notamment sur le temps d'arrêt du système) et sur l'occupation mémoire. La consommation d'énergie diminue puisque la taille des paquets à transmettre diminue elle aussi.

Tissage d'aspects. Les aspects évolution sont encapsulés dans des composants dédiés au chargement de nouveaux composants. Dans [Shen 2007] les deux aspects sont pourtant entremêlés : d'abord l'invocation des méthodes se fait via un composant central qui gère également l'installation et l'activation des composants ; d'un autre coté, comme cela a été dit précédemment, leur IdE est dépendante de l'architecture matérielle.

Sûreté. Ce critère n'est pas considéré dans les travaux référencés.

4.4.2.3 Modification du binaire

Les approches du type *linkage* dynamique (tant *in-site* comme *off-site*) imposent aux développeurs la définition à la conception des frontières des modules fonctio-

nnels ; ces modules seront à posteriori l'objet de l'évolution à l'exécution. Cela pose, comme on a vu précédemment, le problème de la granularité de l'évolution, où une modification mineure entraîne souvent le remplacement de tout un module.

Un certain nombre de techniques permettent la modification du binaire à une granularité très fine, même au niveau de l'octet. Elles se basent sur les générateurs de deltas entre deux fichiers [Baker 1999, Tridgell 2000, Percival 2003], en l'occurrence entre deux binaires exécutables : le premier est celui qui représente le système en exécution, le deuxième est le binaire *après* que les modifications ont été intégrées et que la nouvelle base de code source a été compilée.

Ces techniques nécessitent des informations détaillées sur le déploiement du logiciel dans la mémoire physique. Par rapport à notre boucle de contrôle d'évolution, l'état du système est constitué par l'ensemble des données et des fonctions, et les actions concernent des changements à ce niveau de granularité.

Les travaux de Reijers et Langendoen [Reijers 2003] et de Jeong et Culler [Jeong 2004] ont soulevé une difficulté récurrente de ces approches : des modifications minimes sur le modèle de conception du système (dans tous ces cas, son code source) peuvent produire des deltas volumineux et donc une quantité importante de données à transmettre. Par exemple, l'insertion d'une ligne de code source peut entraîner l'insertion d'un ensemble d'instructions binaires à une adresse spécifique, ce qui à son tour provoque le décalage de toutes les instructions et données situées à une adresse mémoire supérieure. Cette difficulté a des effets négatifs sur la consommation d'énergie et le temps consommé par les actions d'évolution dans le dispositif. Elle trouve son origine dans le peu de contrôle que ces approches ont sur les activités du cycle de vie antérieures à l'exécution, notamment sur la compilation et le déploiement du logiciel.

Des travaux postérieurs ont ciblé cet inconvénient de diverses manières. Koshy et Pandey [Koshy 2005] prennent partiellement en compte la structure du logiciel et, au moment du déploiement original, ils réservent un espace vide en mémoire après le code binaire de chacune des fonctions, en prévision des actions d'évolution qui étendent la zone mémoire allouée à celles-ci [3] ; cela impacte l'occupation mémoire du logiciel. Von Platen et Eker [von Platen 2006] enrichissent l'étape de *linkage* avec des informations sur le placement en mémoire des objets binaires : en laissant des plages de mémoire vides, ils réduisent le nombre de segments à déplacer lors des actions d'évolution.

Ksplice [Arnold 2009] insère des instruction du type *jump* au début des fonctions qui sont remplacées, créant ainsi, de façon dynamique, des indirections vers les nouvelles versions de ces fonctions. Cette technique est utilisée commercialement comme une extension du noyau Linux pour intégrer des patchs pendant son exécution [Ksplice 2010]. Finalement, Zephyr [Panta 2009] introduit des indirections semblables à celle de Ksplice, mais à des étapes antérieures du développement (compilation et déploiement). Il construit une table extensible d'indirections qui est déployée avec le système.

3. Cette approche exploite le phénomène décrit par la deuxième des lois d'évolution de Lehman [Lehman 1985], qui stipule qu'un système devient progressivement plus complexe lors qu'il évolue.

Richesse d'évolution. Les alternatives d'évolution sont nombreuses, dans la mesure où les objets sur lesquels les actions d'évolution opèrent sont d'une granularité très fine. Dans la plus part ces objets sont des fonctions. La seule limite en granularité de modification est fixée par le matériel et la gestion de la mémoire physique.

Impact sur la performance. Il existe un compromis entre consommation d'énergie due à la transmission des deltas de code binaire, et l'occupation mémoire du logiciel. L'impact sur ce dernier critère peut s'avérer important. Les IdE embarquées sont minimales, réduites à un chargeur de code binaire dans la mémoire, puisque la plupart du travail lié à l'évolution est mené *off-line*.

Tissage d'aspects. Le fait qu'une bonne partie des activités d'évolution s'exécutent *off-site* crée une forte dépendance entre l'outillage des IdE et les architectures matérielles sous-jacentes. L'inconvénient majeur des approches basées sur la modification du code binaire est l'extensibilité de l'approche. En effet, ces techniques impliquent des changements parfois importantes sur la manière dont la compilation et le déploiement du code sont faits, et donc sur l'outillage propre au développement logiciel.

Sûreté. L'évaluation de cet aspect varie en fonction des techniques évaluées. Dans [Reijers 2003, Jeong 2004, Koshy 2005] ce critère n'est pas pris en considération. Dans [von Platen 2006, Panta 2009] le système doit se réinitialiser avant et après l'exécution des actions d'évolution, ce qui augmente son temps d'arrêt. Un des objectifs de Ksplice [Arnold 2009] est justement d'appliquer des *patch* sur un noyau Linux sans avoir besoin de le redémarrer ; cette fonctionnalité repose sur une connaissance approfondie du logiciel en exécution et non seulement de son code binaire, ce qui fait que ces techniques soient difficilement extensibles.

4.5 Synthèse sur l'évolution

Les tableaux 4.1 et 4.2 synthétisent les propriétés des catégories de techniques précédemment décrites, par rapport aux critères de performance définis au chapitre 3.2, dénotés :

- RE : Richesse d'évolution
- IP : Impact sur la performance du logiciel
- TA : Tissage d'aspects
- S : Sûreté

Pour chaque critère nous assignons de manière subjective une note entre 1 et 10 afin de placer une comparaison d'ordre qualitatif sur un plan quantitatif.

Concernant les critères tissage d'aspects et sûreté, nous constatons que les techniques qui font usage des modèles architecturaux comme les composants logiciels obtiennent des résultats plutôt satisfaisants. C'est le cas d'AADL [Borde 2009], Fractal/Think [Polakovic 2006] et, dans un moindre degré, TinyOS [Munawar 2010]. Ces

		Modes d'opération – [Borde 2009]	Note
RE	-	Limitée aux modes d'opération définis à la conception	3
	+		
IP	-	Duplication de code, politique d'évolution embarquée	5
	+	Optimisations possibles au niveau de l'IdE (« automatas » et enveloppes des composants)	
TA	-		9
	+	Aspect évolution encapsulé et généré dynamiquement à la compilation	
S	-		9
	+	Haute fiabilité des transitions entre modes, *downtime* minime	
		Total	**6,5**

		Intergiciels – [Matthys 2009]	Note
RE	-	Limitée à celles dictées par les politiques définies à la conception	6
	+	Certaines IdE permettent l'installation de nouvelles politiques et donc d'étendre les possibilités d'évolution	
IP	-	La vérification récurrente des conditions de politiques perturbe le flux d'exécution	2
	+		
TA	-	Modification opaque du comportement du système. Imposition de modes d'exécution	3
	+		
S	-	Pas considéré, implicitement pris en compte par la politique, donc fixé à la conception	5
	+		
		Total	**4**

		Points fixes d'intervention – [Anne 2009]	Note
RE	-		9
	+	Tous les composants offrent des interfaces orientées évolution, toutes les entités du modèle à composants sont modifiables	
IP	-	Surcoût important en termes d'occupation mémoire du à l'inclusion d'interfaces d'introspection et intercession	3
	+	Pas d'impact sur les temps d'exécution lors d'un état « stable ». Faible temps consommé par les actions d'évolution	
TA	-	Une partie du raisonnement est fait dans les composants fonctionnels, ce qui affecte leur réutilisation	6
	+	Le tissage à la conception peut être améliore par des bonnes pratiques de conception	
S	-		9
	+	Techniques assurant la fiabilité des actions d'évolution et un *downtime* minime	
		Total	**6,75**

TABLE 4.1 – Synthèse des techniques qui considèrent l'évolution au niveau des composants

		***Linkage* Dynamique**	Note
RE	-	L'objet de l'évolution est le composant, dont la granularité est définie à la conception	6
	+	Tous les composants sont remplaçables	
IP	-	Compromis entre granularité des composants et performance au niveau de l'occupation mémoire et la consommation d'énergie lors de la transmission des données	4
	+	Pas d'impact sur les temps d'exécution lors d'un état « stable ». Faible temps consommé par les actions d'évolution	
TA	-		8
	+	Aspects évolution encapsulés par des modules dédiés, chargés à la compilation	
S	-	Exécution du système arrêtée lors des activités d'évolution	5
	+	-	
		Total	**5,75**

		Pre-*linkage off-site*	Note
RE	-	L'objet de l'évolution est le composant, dont la granularité est définie à la conception	6
	+	Tous les composants sont remplaçables	
IP	-		7
	+	Par rapport au *linkage* dynamique, l'occupation mémoire et la consommation d'énergie diminuent	
TA	-	Dépendance vis-à-vis les architectures matérielles	6
	+	Aspects "évolution" encapsulés par des modules dédiés chargés à la compilation.	
S	-	Application arrêtée lors des activités d'évolution	5
	+		
		Total	**6,25**

		Modification binaire	Note
RE	-		9
	+	Objets d'évolution de granularité très fine	
IP	-	Compromis entre consommation d'énergie et occupation mémoire	7
	+	IdE embarquées minimales	
TA	-	Dépendance forte vis-à-vis les architectures matérielles et logicielles. Approche peu extensible.	4
	+	Aspects "évolution" encapsulés par des modules dédiés chargés à la compilation	
S	-	Application arrêtée lors des activités d'évolution	5
	+	-	
		Total	**6,25**

TABLE 4.2 – Synthèse des techniques qui considèrent l'évolution en dehors du composant

techniques font partie des familles Modes d'opération, Points fixes d'évolution et Linkage dynamique, respectivement. La structuration du logiciel par composants favorise clairement la séparation des préoccupations fonctionnelles et extra-fonctionnelles des la conception jusqu'à l'exécution, contribuant ainsi à l'amélioration des performances des IdE.

D'ailleurs, ce constat est confirmé par les résultats obtenus par Oreizy et Taylor dans [Oreizy 1998, Oreizy 2008]. Ils ont montré le rôle bénéfique qu'ont les modèles architecturaux (dont les composants logiciels étudiés au chapitre 2.2 font partie), et en particulier celui des connecteurs, au moment de l'implémentation et de l'exécution des actions d'évolution. Leurs travaux concernent le domaine du logiciel au sens large et non spécifiquement le domaine de l'embarqué.

Néanmoins, concernant ce même ensemble de techniques, nous constatons l'existence d'un **fort compromis entre la richesse d'évolution et l'impact des IdE sur la performance du logiciel**. Dans AADL [Borde 2009] les modes d'opération, qui déterminent les alternatives de changement à l'exécution, sont figés à la conception du système ; cela leur permet d'optimiser les transitions des modes et minimiser l'IdE embarquée. Au contraire, dans [Polakovic 2006, Munawar 2010] tous les composants sont plausibles d'évoluer des diverses manières, indépendamment de leur granularité ; cela demande une IdE embarquée volumineuse. Dans Contiki [Dunkels 2006] et Dynamic TinyOS [Munawar 2010], des modules de granularité plus fine induisent une occupation mémoire supérieure.

En contraste, les techniques dites « tardives » telles que le Pre-*linkage* et spécialement la Modification du binaire [Arnold 2009, Panta 2009, Guo 2008, von Platen 2006] ne semblent pas posséder cette caractéristique. Elles supportent une granularité d'évolution très fine sans trop de pertes relatives à la performance du système. Cela se fait par l'utilisation des informations concernant l'exécution et le déploiement du logiciel dans les activités propres à l'évolution. Néanmoins, au fur et à mesure que le volume d'information augmente, la dépendance vis-à-vis les plates-formes matérielles et logicielles augmente elle aussi, ce qui affecte l'extensibilité de ces techniques.

Nous considérons que ce phénomène n'a pas été suffisamment étudié, particulièrement par les techniques de modification du code binaire, en grande partie parce qu'elles manquent des modèles architecturaux et comportementaux de haut-niveau leur permettant d'abstraire ou de mieux analyser des aspects liés aux plates-formes de bas niveau. On ne retrouve que dans [Guo 2008] l'utilisation d'un modèle des dépendances avec les composants déployés.

4.6 Synthèse de la première partie de la thèse

Le domaine de cette étude est le développement des systèmes embarqués, i.e. des logiciels destinés s'exécuter sur une plate-forme matérielle embarqué dans un dispositif et qui effectuent des calculs liés à des processus physiques. Au chapitre 2.1 nous avons identifié les caractéristiques de ces systèmes, dont nous considérons plus particulièrement deux ; ainsi, nous nous intéressons aux systèmes avec des *fortes contraintes* au niveau de l'utilisation des ressources physiques limitées, et très *ouverts*

au changement.

Les modèles à base de composants logiciels ont été introduits au chapitre 2.2 de cette thèse. Ces modèles de conception logicielle à haut niveau d'abstraction représentent une solution vis-à-vis des préoccupations d'ordre économique inhérentes au développement logiciel, qui ne sont pas étudiées dans cette thèse mais qui ont une importance notable. Par conséquent, nous nous intéressons dans cette thèse aux plates-formes de développement de systèmes embarqués basées sur les modèles de composants logiciels, et concrètement sur celles où le comportement des composants est décrit en langage C.

La question qui est posée dans cette thèse, à laquelle on essaie de répondre aux chapitres suivants, est la suivante : **comment les composants logiciels doivent être développés de façon à satisfaire les exigences qui se dérivent des fortes contraintes physiques et de l'ouverture au changement ?** Tout d'abord, pour répondre à cette question, il est nécessaire de caractériser le processus de développement des composants : au chapitre 2.2.3 nous avons modélisé ce processus en cinq *étapes* du cycle de vie du logiciel, qui peuvent être englobées en trois *phases* : conception, mise en œuvre et utilisation des composants.

Ensuite, nous constatons que cette question concerne la façon dont le processus de développement est impacté par les deux aspects extra-fonctionnels considérés. Concernant les contraintes physiques, des techniques d'*optimisation* du logiciel ont été proposées ces dernières années. Également, en ce qui concerne l'ouverture au changement des techniques d'*évolution* du logiciel à l'exécution ont fait surface. Nous avons étudié ces deux aspects, respectivement *et de manière indépendante*, aux chapitres 3 et 4 de cette thèse, où nous avons également présenté les références bibliographiques des techniques d'optimisation et d'évolution existantes.

Nous présentons maintenant les principales idées retenues de l'étude de l'état de l'art concernant ces deux aspects :

Optimisation. Nous nous focalisons sur trois critères de performance du logiciel : l'occupation mémoire, le temps d'exécution et la consommation d'énergie en un degré inférieur. A travers de l'étude réalisée nous avons constaté une très forte corrélation entre l'étape du cycle de vie où l'optimisation est effectuée et les ressources consommées dans le processus d'optimisation : plus on attend pour optimiser, plus les ressources consommées sont importantes.

Puisque notre domaine d'étude est caractérisé par les fortes contraintes au niveau des ressources et s'intéresse plus particulièrement à la structure du système, nous avons intérêt à optimiser le logiciel à la compilation, et plus précisément à l'étape de génération de code C.

D'ailleurs, on constate que celui-ci est le choix pris par les plates-formes de développement basées sur des composants logiciels pour l'embarqué. Les plates-formes évaluées effectuent des optimisations à la compilation du logiciel, se focalisant majoritairement sur l'optimisation de la « glue » qui assure l'interopérabilité des composants issus de la conception, et qui est générée par les chaînes de compilation.

Évolution. Concernant l'évolution du logiciel à l'exécution, nous constatons que les plates-formes de développement basées sur les modèles de composants logiciels

ont été de façon générale bien évaluées par rapport aux quatre critères choisis : richesse d'évolution, impact de l'évolution sur la performance du logiciel, séparation des préoccupations et fiabilité de l'évolution.

Nous confirmons ainsi les hypothèses de Oreizy et Taylor [Oreizy 1998] selon lesquelles les modèles architecturaux et notamment ceux basés sur des composants logiciels jouent un rôle bénéfique dans l'implémentation des techniques d'évolution du logiciel.

Le coeur de la question posée dans cette thèse ne se trouve pas autant dans l'étude indépendante de ces deux aspects extra-fonctionnels du logiciel embarqué, mais justement dans l'étude de l'entrelacement entre ceux-ci. En effet, nous avons constaté un **conflit entre la richesse d'évolution du logiciel et l'utilisation économe des ressources physiques du système**. Ce conflit est particulièrement critique dans le cadre de notre étude, puisqu'il est directement lié à l'existence des fortes contraintes matérielles et à l'ouverture au changement, caractéristiques évoquées au chapitre 2 :

- Les plates-formes de développement de composants embarqués appliquent des optimisations de manière systématique, cherchant la minimisation de l'occupation mémoire et du temps d'exécution. Ces optimisation agressives limitent en grande mesure les possibilités d'évolution du système après son déploiement.
- Les logiciels embarqués conçus à base de modèles à composants, qui exhibent une richesse d'évolution significative, doivent embarquer dans la mémoire du dispositif, outre le données et le code fonctionnels, les infrastructures d'évolution (IdE) permettant l'execution des actions d'évolution prévues. La taille de cette IdE impacte l'occupation mémoire du système.

Suite à notre étude, nous constatons qu'aucune des plates-formes de développement de systèmes embarqués basées sur des composants logiciels ne traite, de manière conjointe, les problématiques d'optimisation et d'évolution du logiciel comme elles ont été définies aux chapitres 3 et 4. La Partie II de cette thèse, composée des chapitres 5, 6 et 7, est consacrée à la recherche d'un processus de développement logiciel qui satisfasse, au même temps, les exigences d'optimisation et d'évolution.

Nous identifions un ensemble de directions qui guident notre démarche, issues de l'étude réalisée dans cette première partie de la thèse.

- Le processus de développement des composants doit avoir le minimum de dépendances vis-à-vis des plates-formes matérielles et/ou logicielles sous-jacentes, des modèles d'exécution du logiciel et des fonctionnalités des applications visées. Pour autant, il doit être compatible avec les procédures industriels existantes, concrètement au niveau de la réutilisation de code *legacy* en langage C pour décrire le comportement des composants.
- Étant donné ces restrictions, les techniques d'optimisation implémentées se focalisent sur les assemblages de composants, i.e. sur la structure du logiciel,

et non sur le comportement désiré des composants. Celle-ci est aussi la démarche choisie par les plates-formes de développement étudiées tout au long de cette partie de la thèse, où les optimisations se concentrent sur la génération automatique de code lors de la compilation du système. Les optimisations menées sont assez agressives, afin de combler avec les exigences au niveau de l'utilisation de ressources physiques.

- Afin de réconcilier la richesse d'évolution avec la performance du logiciel, le processus de développement devra permettre un contrôle du niveau d'optimisation du logiciel, contrairement aux plates-formes évaluées où le logiciel est optimisé agressivement et systématiquement. Également, les techniques d'évolution implémentées devront emprunter des idées d'autres approches non basées sur des composants logiciels, comme ceux où les IdE sont intégrées "tardivement" au système (Pre-*linkage off-site* et Modification du binaire, cf. 4.4.2.2 et 4.4.2.3, respectivement). La taille de ces IdE est nettement inférieur dû à l'utilisation des informations concernant l'exécution et le déploiement du logiciel dans les activités propres à l'évolution.

Dans le cadre de cette thèse nous nous appuyons sur une des plates-formes de développement évaluées. Nous faisons le choix de Fractal/Think [Anne 2009], fondés sur l'accent qui est mis sur le déploiement d'applications administrables. Fractal/Think n'implémente pas pour autant des techniques d'optimisation du logiciel généré, ni des techniques d'évolution qui prennent compte des limitations de ressources, soit l'objet de cette thèse.

Chapitre 5

Réifications de Composants

Sommaire

5.1 Introduction **69**
5.2 Réification **70**
5.2.1 Définition 70
5.2.2 Propriétés 73
5.2.3 Communication entre réifications 74
5.2.4 Composants miroir 75
5.3 Aperçu de la contribution **76**
5.3.1 Réifications pour l'optimisation 77
5.3.2 Réifications pour l'évolution 78

Ce chapitre vise à établir les fondements qui nous permettront, aux chapitres suivants, de proposer un processus de développement de composants logiciels, afin de satisfaire les exigences propres aux systèmes embarqués en termes de contraintes d'utilisation de ressources et de besoin d'ouverture aux changements. Après une introduction au chapitre 5.1, nous définissons et caractérisons la notion de réification au chapitre 5.2 et présentons au chapitre 5.3 un aperçu de la contribution et comment elle utilise les concepts définis.

5.1 Introduction

Jusqu'à ce point de la thèse, nous avons étudié l'optimisation et l'évolution logicielle de manière séparée. Ce choix est justifié par le besoin d'identifier, de définir et d'analyser en détail les préoccupations propres à chacun de ces aspects, ainsi que d'étudier les solutions qui ont été proposées, en particulier celles qui se basent sur l'utilisation de modèles à composants.

Dans l'étude réalisée dans la première partie de cette thèse, nous avons vu que les techniques d'optimisation qui sont implémentées concernent la structure du logiciel, et qu'elles sont de préférence appliquées lors de la *mise en œuvre* du logiciel. D'autre part, pour l'évolution du logiciel lors de son *exécution*, les activités d'évolution bénéficient de l'utilisation d'entités d'un plus haut niveau d'abstraction. Nous avons donc besoin d'une représentation du logiciel qui soit commune à ces deux étapes du cycle de vie. Or nous constatons que nous manquons d'un cadre qui nous permette de considérer au moins ces deux aspects dans un même environneent de modélisation.

Les composants sont des candidats naturels à assumer un rôle tout au long du développement. Pourtant, l'utilisation des composants nécessite un raffinement de

leur définition selon le domaine et la position dans le cycle de vie. Les diverses définitions partagent néanmoins la notion de composant comme une unité d'abstraction de comportements et de données, destinée à être assemblée afin de construire des composants plus complexes.

Nous proposons de lier cette notion de composant à plusieurs représentations. Au chapitre suivant, nous introduisons les *réifications*, qui sont les diverses représentations selon les étapes du cycle de développement des composants définies au chapitre 2.2.3 (cf. figure 2.3 page 17). Ensuite nous décrivons la manière dont les réifications sont utilisées pour satisfaire les exigences propres aux systèmes embarqués évoquées précédemment. Les chapitres 6 et 7 détaillent les solutions apportées.

5.2 Réification

Comme cela a été évoqué au chapitre 2.2.4 de cette thèse, en ce qui concerne la syntaxe utilisé pour représenter les composants, on remarque qu'ils existent de très nombreuses manières de décrire un composant logiciel ou, en autres termes, de "matérialiser" son existence en concrétisant l'idée abstraite de composant dans un langage quelconque. D'ailleurs, celle-ci est une caractéristique propre non seulement aux composants, mais aux systèmes logiciels en général. Ce constat nous amène à proposer la définition suivante de la notion de réification.

5.2.1 Définition

Une *réification* d'un composant est une collection de données et de comportements qui encapsulent un ou plusieurs aspects spécifiques du composant à un moment précis de son processus de développement. Dans le cadre de cette thèse nous considérons des réifications à la fin de chacune des étapes définies au chapitre 2.2.3 (cf. figure 2.3). Nous disons, de plus, qu'une réification est une manifestation du composant à cette étape précise de son cycle de vie.

Afin de clarifier ces propos, nous prenons comme exemple un composant vu à travers du processus de développement propre à Fractal/Think, qui correspond au processus classique illustré dans la figure 3.1. La figure 5.1 présente le composant `sound_driver`, qui offre les interfaces `lcc` et `sound` qui encapsulent des services de gestion du pilote et de generation de son. Il requiert les interfaces `lock`, `delay` et `irq`, qui représentent des services de sections critiques, mise en attente et gestion d'interruptions, typiquement offerts par une couche logicielle d'exécutif. Enfin, le composant `sound_driver` possède un attribut `MIN_BEEP_TIME` qui represente la longueur minimale d'un son émis par le périphérique, en ms.

A la *conception*, le composant `sound_driver` est représenté, suivant les conventions définies par le modèle Fractal/Think, par une description de son architecture via un fichier ADL et la définition des types d'interfaces offertes et requises par le composant, via des fichiers IDL. Ces deux types de documents fournissent une description de type « boite noire » ; puisque les contrats définis par les interfaces Fractal/Think sont du type syntaxique (cf. 2.2.4.1 page 20), cette vision « boite noire » ne suffit pas à décrire de façon explicite les aspects fonctionnels et extra-

FIGURE 5.1 – Vue graphique du composant `sound_driver`

fonctionnels du composant. Le comportement du composant, exprimé en langage C, vient compléter cette description.

La figure 5.2 présente le fichier ADL qui correspond au composant `sound_driver`, la définition du type d'interface `bsp.api.sound` de l'interface serveur `sound` via l'IDL Fractal/Think, et un extrait du fichier C qui décrit le comportement du composant, plus concrètement la définition de la méthode `beep_async` de l'interface `sound`.

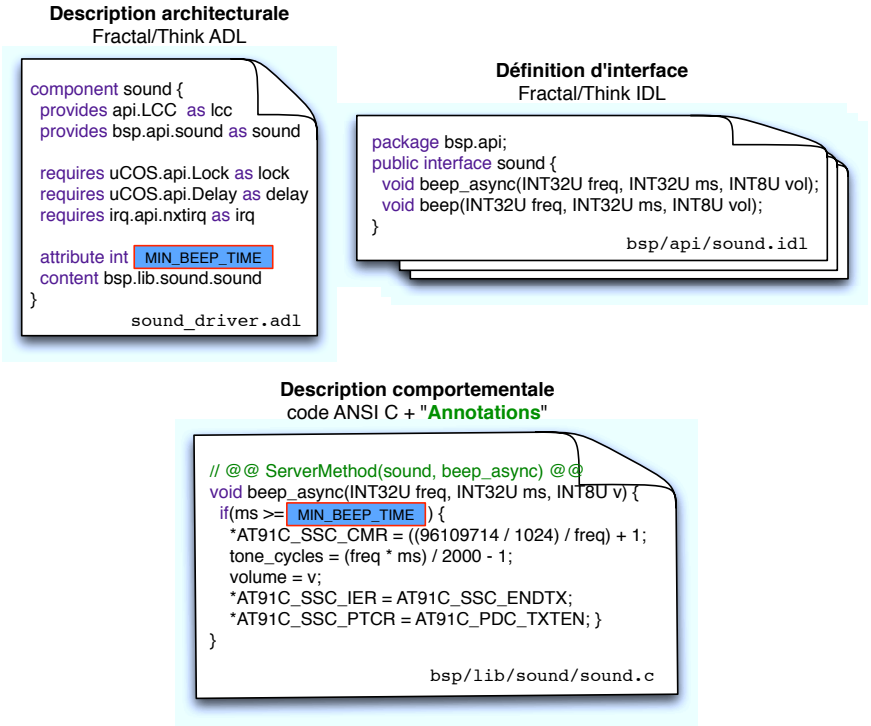


FIGURE 5.2 – Représentation du composant `sound_driver` à la conception. Fichiers ADL, IDL et C (extrait).

Une *instance* (cf. 6.7.1) de ce composant `sound_driver` est inclut dans un assemblage de composants ou composite. Dans notre exemple, il est inclus dans le composite `bsp`, au sein duquel on lui donne le nom `sound`. Le composite `bsp` est à son tour inclus dans le composite `simple`, qui représente tout le système.

Lors d'une première phase de *compilation*, le compilateur Fractal/Think génère du code ANSI C pour le composant `sound_driver`, à partir de sa description de haut niveau et de l'architecture globale du système. La figure 5.3 présente un extrait du code généré, celui qui correspond à la définition de la méthode `beep_async` de l'interface `sound`. Il est possible de retrouver les entités du modèle à composants dans le code généré par la chaîne de compilation. Dans l'exemple, nous retrouvons des références à l'attribut `MIN_BEEP_TIME` et à l'interface serveur `sound`, définis à la conception.

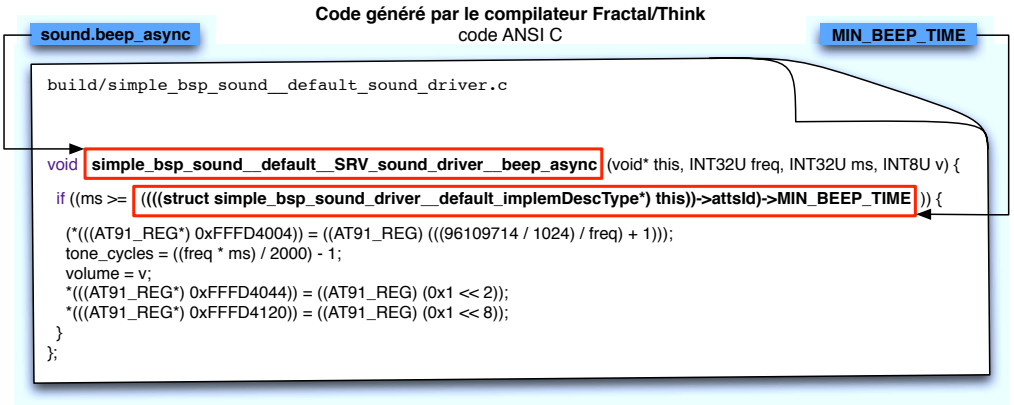


FIGURE 5.3 – Représentation du composant `sound_driver` après une premiere passe de compilation Fractal/Think. Extrait du code C généré. Il est possible d'identifier des correspondances entre certaines abstractions utilisées à la conception et le code C généré, comme pour l'attribut `MIN_BEEP_TIME` et la méthode `beep_async` de l'interface `sound`.

Le code généré par Fractal/Think est ensuite compilé par GCC afin d'obtenir le code binaire qui est interprété par un processeur. Les fonctionnalités et données propres au composant, ainsi que les abstractions utilisées pour faciliter la conception du système, sont maintenant, soit disparus, soit représentés dans un langage binaire.

La figure 5.4 présente un extrait du code binaire en assembleur ARM7. Il est possible, sous certaines conditions qui seront étudiées aux chapitres suivants, de tracer une correspondance entre les abstractions utilisées lors de la conception du système et le code binaire en exécution. Tel est le cas de l'attribut `MIN_BEEP_TIME` du composant `sound_driver` dans la figure 5.4. La valeur de cet attribut à l'exécution est stockée dans une casse mémoire de longueur 32 bits à l'adresse `0x2004dc`.

Afin de réifier une référence à la valeur de cet attribut à l'exécution comme celle de la ligne de code `if (ms >= MIN_BEEP_TIME)` dans la figure 5.2, la valeur stockée dans `0x2004dc` est récupérée. Le code binaire `ldr r3, [r0, #32]` (a) copie dans le registre `r3` la valeur stockée dans l'adresse mémoire `r0 + 32`, où est stockée la valeur de l'attribut. Ensuite, `r3` est comparé à `r4`, qui contient la valeur du deuxième paramètre de la fonction `beep_async`, `ms`. La comparaison dans `if (ms >= MIN_BEEP_TIME)` est ainsi réifiée en assembleur.

Les trois représentations du composant `sound_driver` à la conception, à la com-

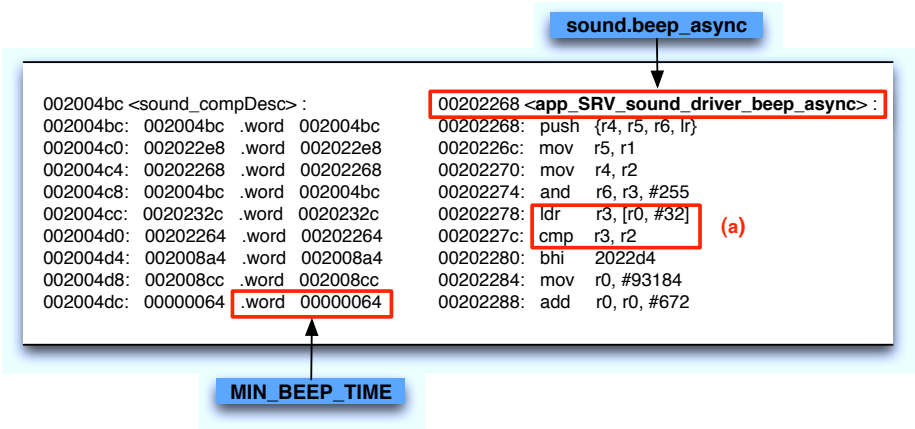


FIGURE 5.4 – Représentation du composant `sound_driver` à l'exécution. Extrait du code binaire. Il est possible d'identifier des correspondances entre certaines abstractions utilisées à la conception et les données du fichier exécutable, comme pour l'attribut `MIN_BEEP_TIME`

pilation et à l'exécution, illustrées dans les figures 5.2, 5.3 et 5.4, respectivement, font référence à la même instance du composant `sound_driver` à des étapes distinctes de son cycle de vie. Elles sont donc des réifications de ce composant, même si le langage utilisé pour les représenter est différente.

5.2.2 Propriétés

Une manière de caractériser les réifications est de lister les propriétés qui les caractérisent. Nous rappelons qu'une réification est une vue, qui peut être incomplète, d'un seul et même composant. Dans ce cadre, par rapport aux réifications nous définissons l'ensemble de propriétés suivant :

i. Relation d'application : une réification est liée à une unique étape du cycle de vie du composant.

ii. Non-injection : plusieurs réifications peuvent être liées à une même étape du cycle de vie. On exige de ces réifications qu'elles soient équivalentes vis-à-vis les aspects qui sont encapsulées par elles.

iii. Aucune restriction n'est imposée au niveau du langage utilisé pour décrire une réification.

iv. Une réification est crée à l'étape S du cycle de vie des composants à laquelle elle fait référence. Son existence n'est pourtant pas restreinte à cette étape et nous n'imposons pas de limite temporelle à ce sujet.

v. Si une réification encapsule des comportements, on dit que cette réification est *exécutable*.

   a. De la propriété ii, on en déduit que lorsque l'exécution d'une réification modifie son état, toutes les réifications qui font référence au même stade du cycle de vie devront être modifiées pour assurer la consistance entre les réifications.

b. De la propriété iii, on déduit qu'une réification exécutable peut être décrite par les mêmes éléments architecturaux et abstractions propres aux modèles à composants. Une réification peut être donc considérée comme un composant logiciel.

Ces propriétés établissent un cadre de travail par rapport aux réifications, déterminant la portée de l'utilisation que l'on peut en faire. En particulier, elles déterminent la nature des communications entre réifications, qui seront étudiées au chapitre suivant.

5.2.3 Communication entre réifications

Une réification peut interagir avec d'autres réifications en établissant des liens de trois types, qui sont illustrés par la figure 5.5 où des réifications de deux composants A et B aux étapes de compilation et de déploiement communiquent entre elles par des liens verticaux (1), horizontaux (2) et internes (3).

Puisque nous ne faisons aucune hypothèse concernant l'implémentation ou le modèle d'exécution des réifications, les définitions des liens entre elles présentées par la suite restent assez générales. Elles seront raffinées lors des chapitres suivants.

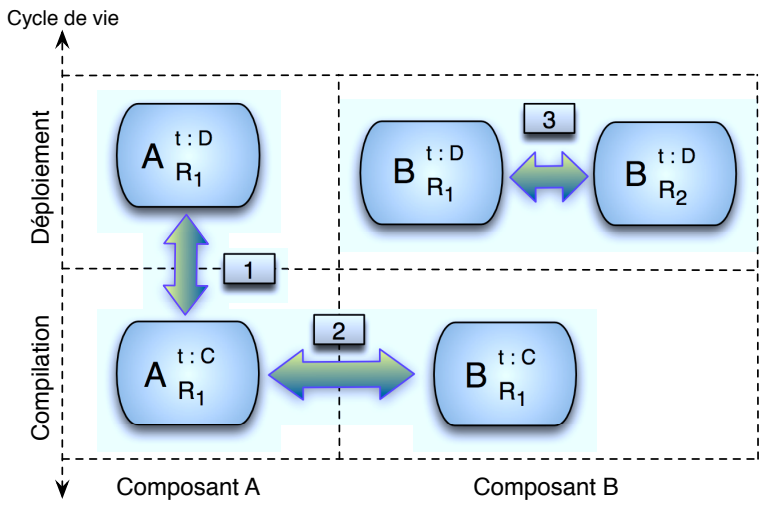


FIGURE 5.5 – Illustration des liens verticaux (1), horizontaux (2) et internes (3) entre réifications des composants A et B référençant les étapes de Compilation (C) et Déploiement (D). $X_y^{t:T}$ représente la réification y du composant X qui fait référence à l'étape T du cycle de vie du composant.

5.2.3.1 Liens Verticaux

Les liens verticaux concernent deux réifications qui font référence à un même composant à des étapes distinctes de son cycle de vie. Ce type de lien établit une relation entre les diverses manières de représenter un composant à travers son cycle de

vie et crée un *continuum* entre les activités propres à chacune des étapes du développement logiciel, qui sont souvent menées de manière isolée. Afin de simplifier notre démarche, nous considérons qu'une réification ne peut établir des liens verticaux qu'avec des réifications qui font référence à une étape immédiatement supérieure ou inférieure du cycle de vie.

Dans Fractal/Think le mécanisme des *propriétés* permet d'intégrer aux descriptions architecturales des préoccupations extra-fonctionnelles comme la réconfiguration dynamique et les modèles d'exécution du logiciel, comme dans [Polakovic 2008] et [Poulhies 2010]. Cela est fait en introduisant à la conception des propriétés qui influencent le processus de compilation des assemblages de composants par le rajout et l'intégration de composants d'une librairie ou en façonnant la génération de code.

Cette manière dont les informations contenues dans les descriptions architecturales des composants à la conception déterminent le processus de compilation de ceux-ci, est un exemple d'un lien vertical, malgré le fait que dans les travaux cités l'idée de réification ne soit pas formalisée.

5.2.3.2 Liens Horizontaux

Ils concernent deux réifications liées à deux composants distincts, mais qui font référence à une même étape du cycle de vie logiciel. Les connexions qui lient deux composants à l'exécution (des appels à procédure classiques dans Fractal/Think) sont des liens de type horizontal. Au chapitre 7.3 nous étendrons cette idée à d'autres étapes du cycle de vie des composants.

5.2.3.3 Liens Internes

Ils concernent deux réifications qui référencent un même composant à une étape de son cycle de vie. Les liens entre la réification qui évolue sur la plate-forme d'exécution et son composant *miroir* (qui est défini au chapitre suivant) sont des exemples des liens de type interne.

5.2.4 Composants miroir

Un cas particulier de réification qui mérite une réflexion supplémentaire est celui des réifications de composants lors de son exécution. Lorsqu'on parle de l'exécution d'un logiciel basé sur des composants, on fait référence à au moins un objet logiciel [1] construit aux étapes du cycle de vie précédentes (par des séquences des liens verticaux) qui s'exécute sur une plate-forme logicielle et/ou matérielle donnée et dont son comportement et ses attributs correspondent à ceux spécifiés lors de la conception du composant auquel il fait référence. Cet objet est une réification à l'exécution de ce composant.

La propriété (ii) indique pourtant que cette réification n'est pas unique. Ainsi, il peut exister une autre réification qui fasse référence ce même composant à l'exécu-

1. A ce point nous encourageons le lecteur à éviter de faire un lien quelconque entre le terme *objet logiciel* ici employé et la programmation orientée objet (*Object-Oriented Programming*, OOP), ainsi qu'entre les réifications et tous les concepts relatifs à la OOP : classes, instances... Ici, le terme objet logiciel a un sens très générique, désignant une entité logicielle.

tion. Si de plus, cette réification est définie elle aussi dans un modèle à composants (et peut donc être considérée comme un composant logiciel), nous l'appelons un *composant miroir*.

Le mot « miroir » peut se prêter à des interprétations erronées ou insuffisantes. Par la propriété (ii), toutes les réifications à l'exécution doivent être consistantes entre elles. Un composant miroir peut donc être considéré comme « l'image » du composant qui évolue sur la plate-forme d'exécution du système, et de cette manière le sens traditionnel du mot miroir est capturé. Néanmoins, dans les chapitres suivants nous exploitons le fait qu'un composant miroir peut aussi évoluer dans un contexte d'opération qui diffère de celui des réifications à l'exécution traditionnelles. Il peut donc traiter des aspects liés au comportement du composant logiciel d'une autre manière [2].

Les réifications, dont les composants miroir, forment les concepts de base de la contribution de cette thèse. Les chapitres suivants, ainsi que les chapitres 6 et 7 de cette thèse, détaillent la manière dont les réifications sont implémentées et comment leur implantation contribue à l'optimisation et à l'évolution des composants logiciels.

5.3 Aperçu de la contribution

L'idée centrale des contributions faites dans cette thèse est d'utiliser les réifications comme abstraction « pivot » afin d' :

- assurer une continuité dans le cycle de vie des composants, ce qui nous permet d'associer les décisions prises par rapport aux deux aspects extra-fonctionnels qui nous intéressent, i.e. l'optimisation de la performance et l'évolution du logiciel, lors du déroulement des activités de développement ;
- utiliser les abstractions propres aux modèles à composants tout au long du cycle de vie, permettant une séparation des préoccupations d'optimisation et d'évolution, ainsi que l'explicitation des dépendances qui existent entre les composants par rapport à ces deux aspects.

Nous proposons un cadre de développement (cf. figure 5.6) où chaque composant, défini à la conception du système, possède au moins une réification qui est créée à chaque étape de son cycle de vie. Les activités de développement se focalisent sur la construction de ces réifications, et pour cela on établit des liens verticaux entre réifications antérieures du cycle de vie du composant. La figure 5.6 présente une version simplifiée de cette approche, dans la mesure ou les réifications du composant `sound_driver` font référence aux phases et non aux étapes du cycle de vie.

Les réifications permettent d'assurer la traçabilité des composants tout au long de leur cycle de vie. La traçabilité des décisions prises dans le processus de développement permet de remonter du code binaire exécutable vers les exigences qui le

2. Dans cet ordre d'idées, le mot « miroir » a ici un sens similaire à celui qu'on lui donne dans certains ouvrages de la littérature et de l'art moderne et contemporain. Dans *Through the Looking-Glass*, Lewis Carrol utilise un miroir pour symboliser le passage entre le monde réel et un autre monde alternatif où des lois différentes s'appliquent

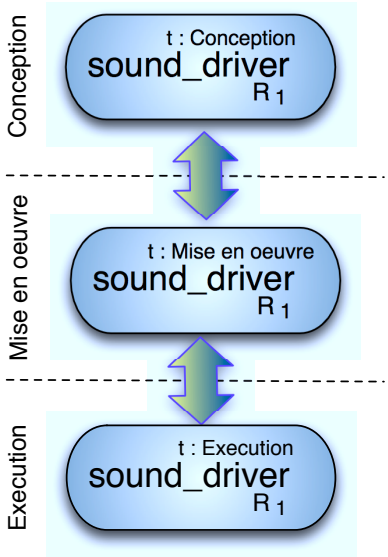


FIGURE 5.6 – Le composant `sound_driver` et ses réifications tout au long des étapes (ici des phases, à des fins de simplification) de son cycle de vie. Des liens verticaux représentent la manière dont les différentes réifications collaborent pour traiter les problématiques d'optimisation et d'évolution.

contraigne ou le justifie, i.e. de l'exécution vers la conception, et vice-versa. Le cycle de vie des composants est donc intégré au modèle de développement.

Dans cette thèse, nous utilisons cette maîtrise du cycle de vie des composants pour proposer des solutions aux problématiques d'optimisation (cf. 5.3.1) et d'évolution (cf. 5.3.2). Ces solutions sont détaillées aux chapitres 6 et 7.

5.3.1 Réifications pour l'optimisation

Les réflexions menées dans la première partie de cette thèse ont permis d'identifier la phase de mise en œuvre comme le moment indiqué pour effectuer des optimisations qui visent l'amélioration des performances du système, par rapport aux contraintes et objectif tracés. En effet, les activités menées lors de cette phase déterminent la manière dont les abstractions propres au modèle à composants sont transformées en code binaire exécutable, ainsi que l'implémentation du code « glue » qui assure l'interopérabilité entre les composants issus de l'étape de conception.

Par exemple, comme illustré par les figures 5.2, 5.3 et 5.4, la valeur de l'attribut `MIN_BEEP_TIME` est tracée jusqu'à une case mémoire spécifique déterminée par les chaînes de compilation et de déploiement du système. De même, l'implémentation de la « glue » entre les entités et entre les composants appartenant à un assemblage quelconque peut varier en fonction de la structuration du logiciel et des caractéristiques des composants, ainsi que des décisions prises lors de la conception du système.

Au chapitre 6 nous nous concentrons sur la compilation des composants, afin de contrôler la génération et la transformation de code source et d'exécuter des

techniques d'optimisation. Nous nous focalisons notamment sur la génération du code « glue ».

L'objectif que nous nous traçons concernant l'optimisation à la compilation des composants Fractal/Think est d'offrir un nombre supérieur de degrés de liberté en ce qui respecte l'optimisation des composants, par rapport à l'état de l'art étudié au chapitre 3. Cette enrichissement d'options d'optimisation est fait sur deux dimensions :

1. Nous proposons d'appliquer des techniques d'optimisation de façon indépendante sur chacune des instances des entités du modèle à composants : attributs, interfaces serveur, interfaces clientes, liaisons, composants primitifs et composites.
2. Nous proposons plusieurs niveaux d'optimisation applicables à chacune des entités de modèle, en opposition aux niveaux optimisé/non-optimisé que l'on observe dans les plates-formes étudiées au chapitre 3. Ces niveaux d'optimisation permettent de régler plus finement la performance du système en fonction de l'ensemble de ses besoins fonctionnels et extra-fonctionnels, parmi lesquels on trouve la capacité d'évolution à l'execution (cf. 5.3.2).

La notion de réification de composant, à travers la possibilité de tracer les instances des entités tout au long du cycle de vie logiciel, est de grande utilité lors de l'optimisation sur les deux dimensions. Concernant la première d'entre elles, elle permet de prendre de décisions sur la génération du code lié à chaque entité de manière indépendante. En ce qui concerne la deuxième, une entité, lorsqu'elle est compilée à deux niveaux différentes d'optimisation, produira deux réifications différentes à la compilation ; ces deux réifications garderont (ou pas) des traces de l'information contenue dans la réification de cette entité à la conception.

Puisque l'implémentation de la « glue » entre entités et entre composants dépend des caractéristiques de plusieurs entités, les dépendances entre les diverses réifications des entités sont un aspect en prendre en compte et qui est également étudié au chapitre 6.

5.3.2 Réifications pour l'évolution

Afin de combler le conflit entre la richesse d'évolution et le coût associé en termes de performance du logiciel, évoqué dans la première partie de cette thèse, la solution proposée se base sur la création d'un composant miroir à l'exécution qui encapsule les fonctionnalités liées à l'évolution du composant.

A différence de la réification traditionnelle du composant à l'exécution, i.e. du code binaire exécutable, le composant miroir n'est pas déployé dans le dispositif embarqué mais dans une machine distante d'administration. Un lien du type interne entre le composant miroir et le composant embarqué assure la consistance entre les deux réifications. De cette manière, les infrastructures d'évolution sont en partie "débarquées" du dispositif, réduisant leur impact sur la performance du logiciel.

Le traitement de l'aspect "évolution", puisqu'il est lié à la dernière étape du cycle de vie du composant (l'exécution), entraîne la participation des réifications liées aux étapes antérieures du cycle de vie. En particulier, les décisions prises par

les réifications de compilation et de déploiement, en particulier en ce qui concerne leur niveau d'optimisation, vont influencer en grande mesure l'exécution des activités d'évolution ainsi que la richesse d'évolution des composants. Nous établissons des liens verticaux entre ces réifications d'un même composant afin de modéliser et traiter ces dépendances.

Le chapitre 7 de cette thèse est entièrement dédié à l'implémentation des infrastructures d'évolution basées sur les réifications des composants aux différentes étapes de leur cycle de vie.

Chapitre 6

Optimisations proposées dans Fractal/Think

Sommaire

6.1 Présentation de la démarche 82
6.2 Attributs 84
6.2.1 Réifications par défaut 84
6.2.2 Coût associé aux réifications 86
6.2.3 Optimisations proposées 87
6.3 Interfaces serveur 89
6.3.1 Réifications par défaut 89
6.3.2 Coût associé aux réifications 91
6.3.3 Optimisations proposées 93
6.4 Interfaces clientes 95
6.4.1 Réifications par défaut 95
6.4.2 Coût associé aux réifications 95
6.4.3 Optimisations proposées 97
6.5 Liaisons 97
6.5.1 Réifications par défaut 97
6.5.2 Coût associé aux réifications 99
6.5.3 Optimisations proposées 101
6.6 Composites 103
6.6.1 Réifications par défaut 103
6.6.2 Coût associé aux réifications 105
6.6.3 Optimisations proposées 107
6.7 Partage de code entre instances de composants 108
6.7.1 Réifications par défaut 108
6.7.2 Coût associé aux réifications 111
6.7.3 Optimisations proposées 111
6.8 Synthèse 112

Dans ce chapitre nous poursuivons deux objectifs. D'une part, nous nous intéressons à caractériser les activités menées par la chaîne de compilation Fractal/Think pour réifier les abstractions propres au modèle à composants utilisées à la conception du logiciel.

D'autre part, nous identifions la façon dont ces réifications peuvent être optimisées et définissons des niveaux d'optimisation pour chaque entité du modèle. Pour

ce faire, nous analysons le processus de construction de réifications à la compilation pour chacune des entités du modèle à composants. Les niveaux d'optimisation proposés seront utilisés lorsque nous nous intéresserons plus en détail aux infrastructures d'évolution du logiciel au chapitre 7.

Nous commençons par préciser rapidement le langage d'entrée du compilateur via un modèle, puis nous détaillons les principes de compilation et d'optimisation.

6.1 Présentation de la démarche

Les entités propres au modèle à composants Fractal/Think constituent notre guide pour cette étude (cf. chapitre 2.2.4.1 de cette thèse). Comme introduction à ce chapitre, nous proposons donc un méta-modèle de Fractal/Think. La figure 6.1 présente une vue simplifiée des entités qui sont définies par la spécification Fractal [Bruneton 2004b].

On retrouve les abstractions évoquées précédemment dans cette thèse : un composant possède un certain nombre d'attributs, expose au moins une interface serveur et requiert un certain nombre d'interfaces clientes. Un composite est un composant, mais il contient en plus des liaisons (associées à une interface cliente et une interface serveur) et au moins un composant. Un contenu est associé à un composant, il contient au moins une définition de méthode.

Les entités dont les réifications à la conception et après la première passe de compilation font l'objet (direct ou indirectement) des chapitres suivants sont encadrées (en gras).

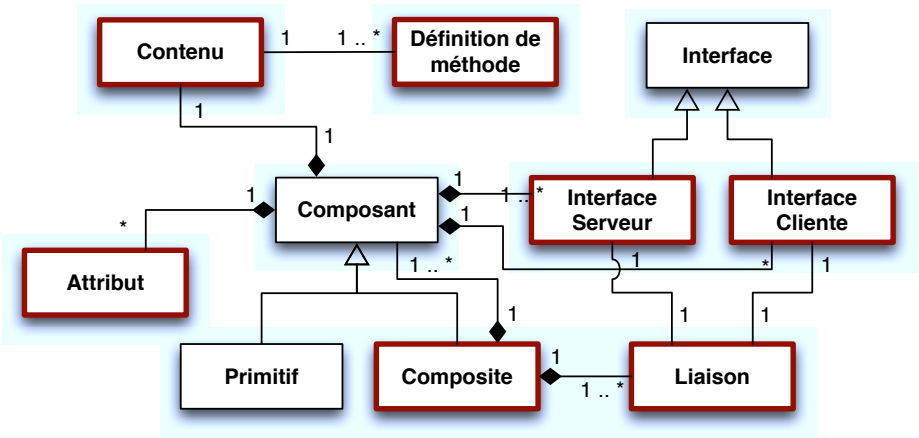


FIGURE 6.1 – Vue des entités du modèle Fractal/Think.

Le code C étant central dans cette étude, la figure 6.2 détaille à son tour la relation qui existe entre le contenu d'un composant (ici décrit en langage C) et les abstractions propres du modèle à composants, notamment les interfaces. Dans la figure 6.2, on se focalise sur les liens entre les définitions des méthodes (serveurs, privées et clientes) et leur invocation en langage C. A ce sujet, il est important

de noter que les appels aux fonctions externes font référence à une déclaration de méthode liée soit à une interface offerte, et donc implémentée par le contenu du même composant, soit requise, et donc implémentée par un autre composant. Les appels aux fonction internes sont implémentées dans le composant lui-même.

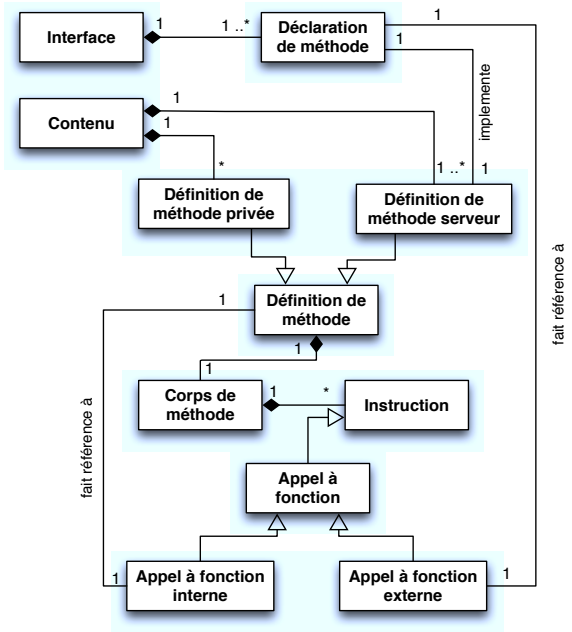


FIGURE 6.2 – Relations existantes entre le contenu des composants (décrit en langage C) et les interfaces du modèle à composants Fractal/Think.

A partir de ces éléments du langage, le processus de compilation dans Fractal/-Think comprend deux étapes. Dans une première étape, les abstractions sont réifiées en code C de deux manières :

- en modifiant le contenu des composants, i.e. le code C fourni par les développeurs, à des points précis et préalablement définis par le modèle à composants lui même ;
- en ajoutant des donnés et du code fonctionnel en C qui sont, soit générés dynamiquement par la chaîne de compilation, soit chargés d'une bibliothèque est adaptés au composant par celle-ci [1].

Ensuite, le code C, résultant de la première étape, est compilé en binaire par un compilateur classique (tel que GCC). Dans ce rapport, nous nous focalisons sur le code C résultant de la première étape de compilation. Mais nous donnons des évaluations en lien avec les opérations du compilateur classique (ici GCC).

1. Une troisième manière consiste en la modification de l'architecture du système au début de sa compilation, en ajoutant des composants dans les assemblages et/ou en ajoutant ou en modifiant les sens des connexions définies par les concepteurs. C'est la démarche suivie dans [Polakovic 2007, Poulhies 2010, Loiret 2010]. Dans cette thèse nous nous focalisons sur les phases ultérieures de la compilation.

Puisque le compilateur Fractal/Think reflète les abstractions du modèle en générant ou en rajoutant du code et des données, les relations entre les abstractions et le contenu à la conception seront également l'objet d'étude de ce chapitre.

Pour chacune des entités du modèle à composants notre présentation se déroule comme suit :

- d'abord nous présentons en détail la manière dont la chaîne de compilation agit par rapport à celles-ci, en particulier comment elle transforme le contenu fourni par les développeurs et/ou génère du nouveau contenu pour le composant ;
- ensuite nous définissons une métrique qui représente le coût (en termes de performance du logiciel) associé à la réification de cette entité ;
- enfin, nous identifions des optimisations possibles et définissons des niveaux d'optimisation pour chacune des entités étudiées.

Nous commençons par les attributs.

6.2 Attributs

6.2.1 Réifications par défaut

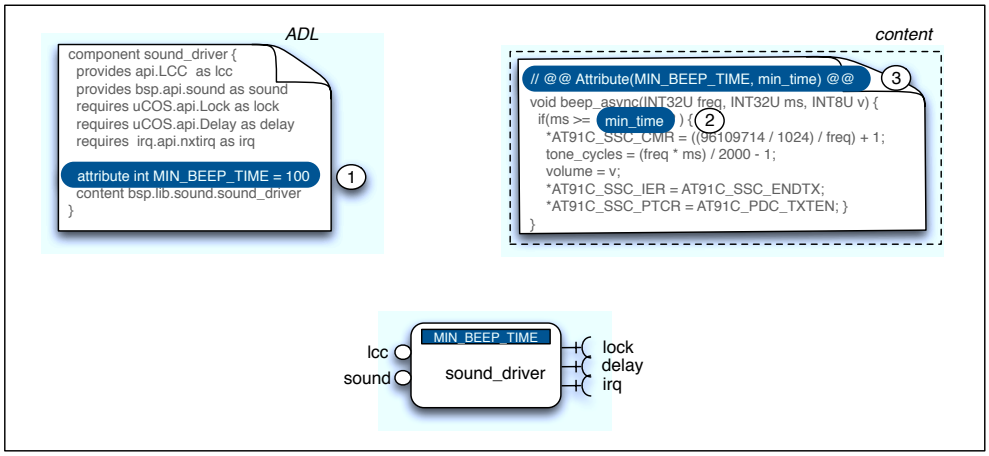
A la conception, les attributs sont déclarés dans le fichier de description du composant (ADL), comme illustré dans la figure 6.3 (1). Ils sont définis par leur type, leur nom et leur valeur initiale (optionnelle). Dans le contenu du composant, la déclaration de l'attribut est supposée déjà faite par ailleurs et il est donc traité comme une variable déjà déclarée (2). La correspondance entre le nom de cette variable et le nom de l'attribut, est établie via une annotation sous forme de commentaire C (3).

La chaîne de compilation standard effectue les actions suivantes pour réifier un attribut :

i. Une structure de données qui stocke la valeur des attributs, `attsDesc`, est créée par la chaîne de compilation (4). Dans le cas où la valeur initiale de l'attribut a été définie à la conception, le membre correspondant est initialisé avec cette valeur. Dans le cas contraire, il est initialisé à une valeur 0.

ii. Toutes les références aux attributs dans le contenu sont transformées en références à la structure de données générée (5). Cela requiert également une référence au descripteur du composant qui est détaillée au chapitre 6.7.

iii. Pour chacun des attributs du composant, des méthodes de consultation (`get`, `list`, `size`) et de modification de la valeur des attributs (`set`) sont ajoutées au contenu du composant (6). Ces méthodes permettent l'accès aux attributs par des composants externes. Elles sont regroupées dans une interface de type `AttributeController` :



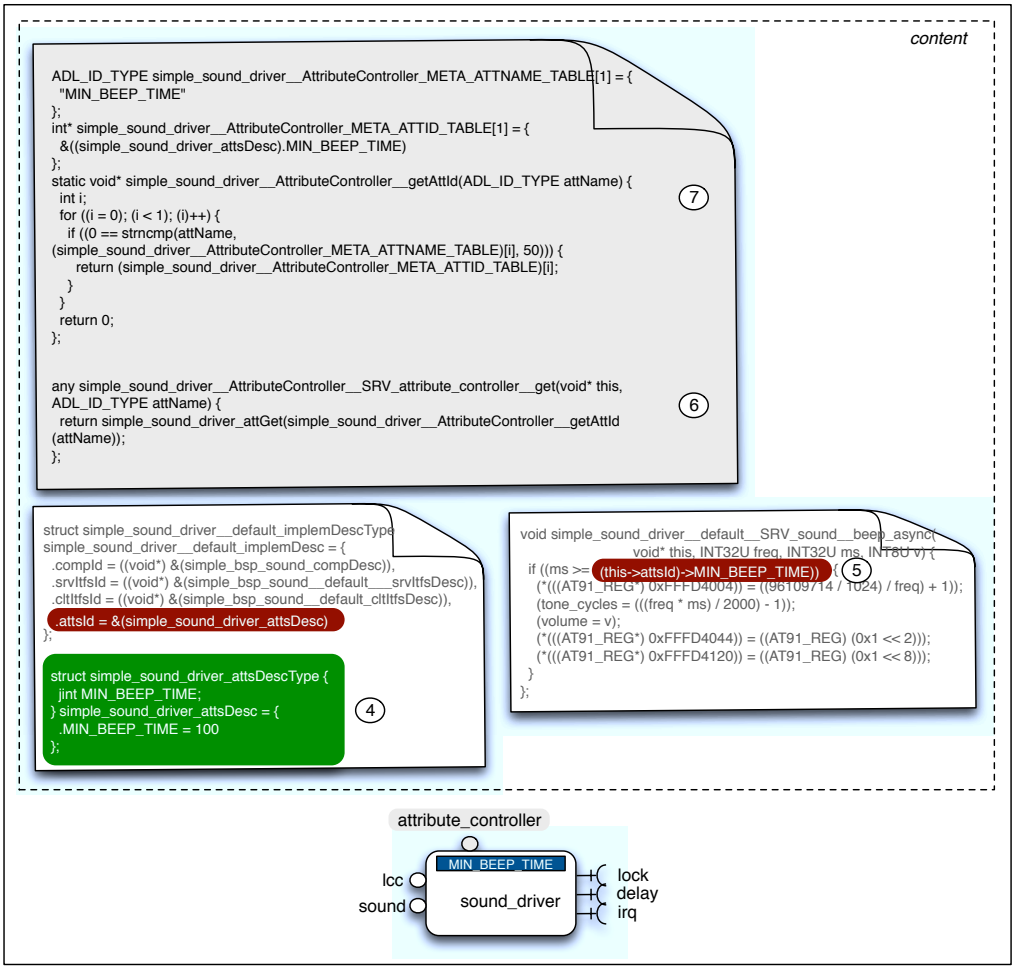


FIGURE 6.3 – Extraits des réifications du composant `sound_driver` à la conception (haut) et après la première phase de compilation (bas). Les éléments liés aux attributs sont mis en relief.

```
public interface AttributeController {
  // set the value of attribute attId
  void set(ADL_ID_TYPE attId, any value);
  // get the value of attribute attId
  any    get(ADL_ID_TYPE attId);
  // get the list of component attributes
  ADL_ID_TYPE* list(void);
  // get the number of attributes
  int    size(void);
}
```

En l'absence de code fourni par les développeurs, la chaîne de compilation génère des implémentations par défaut, ainsi que des méta-données sur les attributs nécessaires à celles-ci. En particulier, concernant la méthode `set`, son implémentation ne prend pas compte des conditions de validité de la valeur de l'attribut. Des structures de méta-données et les méthodes privées nécessaires à l'exécution des méthodes de l'interface `AttributeController` sont également générées et ajoutées au contenu (7).

6.2.2 Coût associé aux réifications

Evaluer le coût associé à la réification d'un seul attribut est difficile, puisque la chaîne de compilation génère des structures de données qui sont en relation avec l'ensemble d'attributs du composant. Nous choisissons d'évaluer le coût associé à l'ensemble d'attributs d'un composant $C(a)$, en termes de l'occupation mémoire due à leurs réifications. Cette même démarche est suivie pour toutes les entités évaluées dans ce chapitre.

Le coût associé aux attributs est donné par :

$$
\begin{aligned}
C(a) &= sizeof(\texttt{attsDesc}) \\
&\quad + sizeof(\texttt{AttributeController}) \\
sizeof(\texttt{attsDesc}) &= \sum_{i}^{N_{attributes}} sizeof(typeof(a_i)) \\
sizeof(\texttt{AttributeController}) &= \sum_{i}^{N_{attributes}} sizeof(typeof(a_i)) \\
&\quad + sizeof(methodesAC) \\
&\quad + (N_{attributes} * sizeof(\texttt{void *})) \\
&\quad + \sum_{i}^{N_{attributes}} sizeof(nameof(a_i))
\end{aligned}
\tag{6.1}
$$

Où :

– $C(a)$: coût associé aux attributs d'un composants, en octets.

- $sizeof$(`attsDesc`) : occupation mémoire de la structure `attsDesc`, qui est la somme de l'occupation mémoire de chacun de ses membres, qui correspondent à chaque attribut du composant.
- $N_{attributes}$: nombre d'attributs du composant.
- $sizeof(typeof(a_i))$: occupation mémoire du membre de `attsDesc` qui correspond à l'attribut i du composant. Le type de ce membre de `attsDesc` est égal à celui de l'attribut (cf. (4) dans la figure 6.3). Pour un attribut de type `int` sur notre plate-forme de référence, cette valeur est de 4 octets.
- $sizeof(methodesAC)$: occupation mémoire des implémentations des méthodes de l'interface `AttributeController` (cf. (6) dans la figure 6.3). Sa valeur est indépendante du nombre d'attributs du composant.
- $sizeof$(`void *`) : occupation mémoire d'un pointeur à un membre de la structure `attsDesc`. Ce pointeur est stocké dans le tableau `META_ATTID_TABLE` (cf. (7) dans la figure 6.3). Le type `void *` est choisi pour abstraire le type de chaque attribut. L'occupation mémoire du pointeur dépend de la place mémoire assignée à un pointeur dans la plate-forme matérielle. Sur 32 bits, elle est de 4 octets.
- $sizeof(nameof(a_i))$: place mémoire nécessaire pour stocker le nom donné à l'attribut i, stocké dans le tableau `META_ATTNAME_TABLE` (cf. (7) dans la figure 6.3).

Exemple. Pour un composant avec un attribut de type `int` et nom `MIN_BEEP_TIME`, en utilisant l'implémentation par défaut fournie par Fractal/Think et compilé par GCC avec un niveau d'optimisation de `-O2`, pour notre plate-forme matérielle de référence, le coût associé à sa réification est de 92 octets, à comparer à 4 octets, taille d'un entier :

$$\begin{aligned} C(a) &= (1 * 4) + (30 + 28 + 8 + 4) + (1 * 4) + 14 \\ &= 92 \text{ octets} \end{aligned}$$

6.2.3 Optimisations proposées

Selon le contexte d'utilisation, il semble opportun de proposer des stratégies d'optimisation pour la génération du code lié à un attribut. Nous définissons 5 niveaux d'optimisation pour la réification d'un attribut d'un composant. Ils sont déduits à partir de la définition des variables suivantes :

- *DESC_ATT* : le membre de la structure de données `attsDesc` correspondant à l'attribut est généré.
- *AC_CONSU* : les méthodes de consultation de l'interface `AttributeController` (`get`, `list`, `size`) sont ajoutées au contenu du composant.
- *AC_MODIF* : la méthode de modification de l'interface `AttributeController` (`set`) est ajoutée au contenu du composant.

Les tableaux 6.1 et 6.2 présentent les valeurs de ces variables pour les cinq niveaux d'optimisation proposés et une description des niveaux. Dans la définition de ces niveaux nous avons pris en compte les dépendances existantes entre les différentes composantes de la réification, et notamment entre l'implémentation de l'interface `AttributeController` et la génération de la structure `attsDesc`.

DESC_ATT	*AC_CONSU*	*AC_MODIF*	**Niveau**
vrai	vrai	vrai	A0
vrai	vrai	faux	A1
vrai	faux	vrai	A2
vrai	faux	faux	A3
faux	faux	faux	A4

Table 6.1 – Variables définies et niveaux d'optimisation des attributs

Niveau	**Description**
A0	Pas d'optimisation. Correspond au niveau par défaut décrit au chapitre 6.2.1.
A1	Seulement les méthodes de consultation (`get`, `list`, `size`) de l'interface `AttributeController` sont ajoutées au contenu du composant. La structure `attsDesc` est obligatoirement générée.
A2	Seulement la méthode de modification (`set`) de l'interface `AttributeController` est ajoutée au contenu du composant. La structure `attsDesc` est obligatoirement générée.
A3	L'interface `AttributeController` n'est pas ajoutée au contenu du composant. Les méta-données associées à cette interface ne sont pas générées. Les attributs restent consultables et modifiables par le composant lui-même grâce à la structure `attsDesc` qui est générée.
A4	Si la valeur de l'attribut est initialisée par défaut, les références à un attribut sont remplacées par cette valeur constante définie à la conception. L'attribut n'est donc plus modifiable. La structure de données `attsDesc`, où la valeur de l'attribut est stockée, n'est pas générée.

Table 6.2 – Description niveaux d'optimisation des attributs

Exemple. Le tableau 6.3 présente le coût associé aux réifications des attributs pour l'exemple au chapitre précèdent. La réification assembleur a été obtenue avec GCC, niveau `-O2`.

Niveau	Coût de réification (octets)
A0	$C(a) = (1 * 4) + (30 + 28 + 8 + 4) + (1 * 4) + 14 = 92$
A1	$C(a) = (1 * 4) + (28 + 8 + 4) + (1 * 4) + 14 = 62$
A2	$C(a) = (1 * 4) + (30) + (1 * 4) + 14 = 38$
A3	$C(a) = (1 * 4) = 4$
A4	$C(a) =$ **0 ou 4**

TABLE 6.3 – Coût de réification d'un attribut entier par rapport au niveaux d'optimisation.

Il est à noter que pour le niveau A4, si la valeur initiale de l'attribut doit être stockée en mémoire, le coût associé est celui d'une variable de type entier (4 octets). Par contre, si la valeur constante est déjà présente dans le binaire, le coût associé est nul. Par rapport au cas où l'attribut est assigné au niveau A3, cette valeur constante peut être stockée et est accessible dans la mémoire non-volatile (ROM ou FLASH).

Considérations sur le compilateur de bas-niveau. On peut noter que le fait de générer une structure pour un attribut même limité à une valeur de type entier (et non directement une variable) limite l'impact du compilateur de bas-niveau. Par exemple, la propagation de constantes (optimisation effectuée par les niveaux `-O2`, `-O3` et `-Os` de GCC) n'est ici pas effective.

D'un autre coté, les propositions faites permettent d'être indépendantes du compilateur choisi et de son utilisation (soit son niveau d'optimisation). Pour autant, on pourrait imaginer pour le niveau A4 de générer une variable initiale et d'utiliser le compilateur GCC avec les options d'optimisation précédemment indiquées pour obtenir un comportement équivalent à A4.

6.3 Interfaces serveur

6.3.1 Réifications par défaut

A la conception, un composant explicite le fait d'offrir une interface via le mot clés `provides` (1, dans la figure 6.4). Le type de l'interface est défini dans un fichier IDL (2). Les méthodes propres à l'interface sont implémentées dans le contenu du composant (3). La correspondance entre le nom de la méthode dans le contenu et le nom de la méthode de l'interface est établie via une annotation sous forme de commentaire (4).

La chaîne de compilation effectue les actions suivantes pour réifier les interfaces serveur :

i. Dans le contenu des composants, les noms des méthodes des interfaces sont modifiés afin de les rendre uniques à l'application (5). Cela se fait en ajoutant comme préfixe une chaîne de caractères unique, construite avec les noms de composants par lesquels, à partir du composant racine on arrive, via composition,

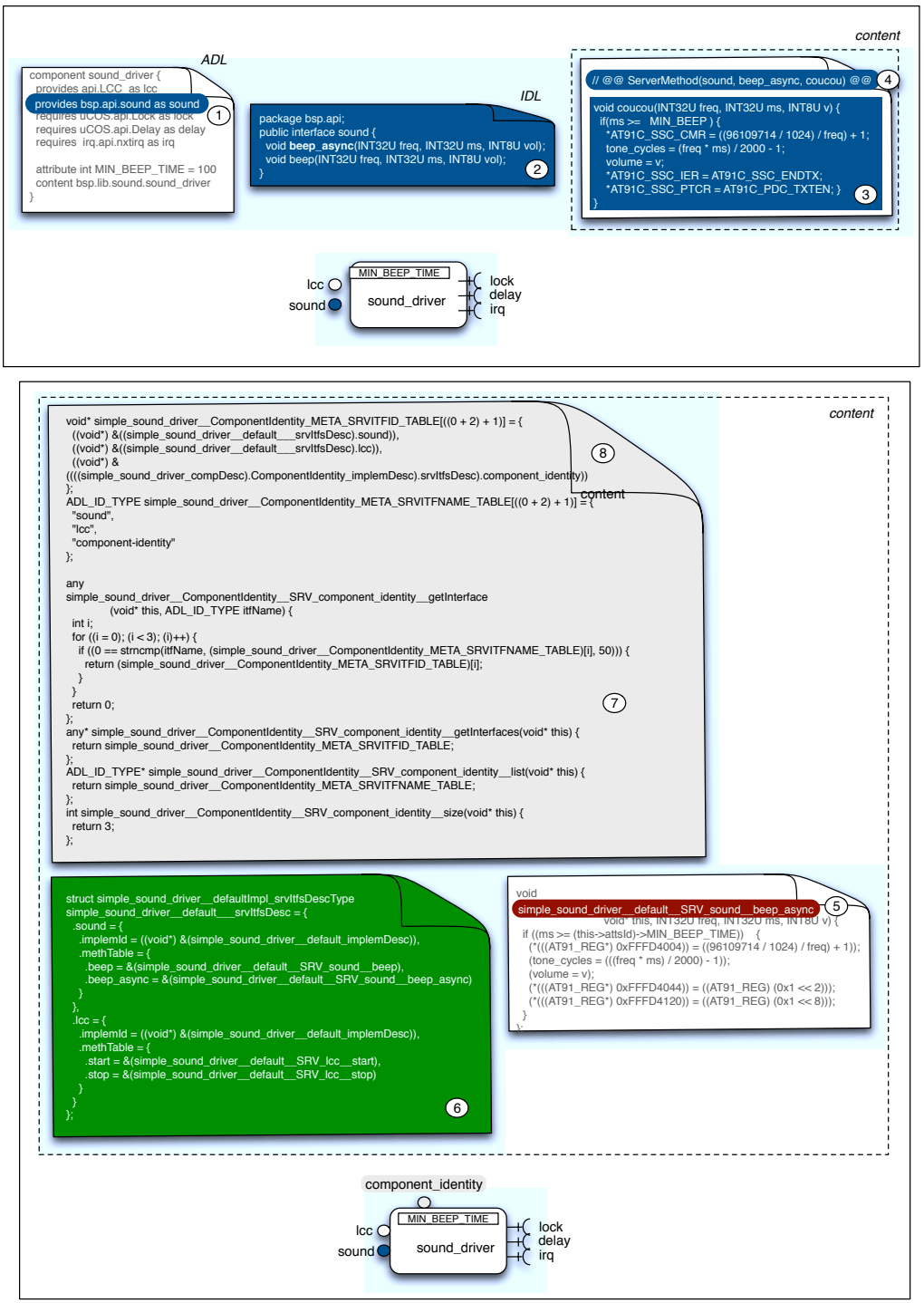


FIGURE 6.4 – Extraits des réifications du composant `sound_driver` à la conception (haut) et après la première phase de compilation (bas). Les éléments liés à l'interface serveur `sound` sont mis en relief.

à celui en question. On ajoute à ce préfixe la chaîne de caractères `__SRV_`*{nom de l'interface}*`__`[2]. Les appels aux fonction internes (i.e. aux méthodes des interfaces serveur ou aux méthodes privées du composant) sont transformées par la chaîne de compilation en vertu de ces règles de nommage.

ii. Une structure de données, `srvItfsDesc`, est générée dynamiquement par la chaîne de compilation (6). Cette structure joue le rôle de descripteur de l'ensemble des interfaces serveur du composant. Chaque membre de la structure joue le rôle de descripteur d'une des interfaces serveur. Il est composé de l'adresse du descripteur du composant (cf. 6.7) et d'une table de méthodes contenant l'adresse de chacune des implémentations des méthodes de l'interface.

iii. Des méthodes permettant aux composants externes d'"introspecter" les interfaces offertes par le composant (leur nom, leur nombre) sont ajoutées au contenu de celui-ci (7). Ces méthodes sont regroupées dans une interface du type
`ComponentIdentity` :

```
public interface ComponentIdentity {
        // get interface from its name
        any      getInterface(ADL_ID_TYPE name) ;
    // get server interfaces
    any* getInterfaces(void) ;
    // get server interfaces list of names
    any* list(void) ;
    // get number of server interfaces
    int size(void);
}
```

En l'absence de code fourni par le développeur, la chaîne de compilation génère des implémentations par défaut. Des méta-données sur les interfaces serveur (8), nécessaires à l'exécution des ces implémentations, sont également générées par la chaîne de compilation.

6.3.2 Coût associé aux réifications

Nous évaluons le coût associé aux interfaces serveur d'un composant $C(is)$, en termes de l'occupation mémoire due à leurs réifications. Ce coût, si l'on utilise l'implémentation par défaut de `ComponentIdentity`, est donné par :

2. Cette technique classique par laquelle on assigne un nom unique à une entité logicielle est appelée *mangling*.

$$
\begin{aligned}
C(is) &= sizeof(\texttt{srvItfsDesc}) \\
&+ sizeof(\texttt{ComponentIdentity}) \\
&+ \sum_{i}^{N_{itfsSrv}} \sum_{j}^{N_{methodes}^{is_i}} sizeof(methode_{ij}) \\
sizeof(\texttt{srvItfsDesc}) &= sizeof(\texttt{void *}) * \left(\sum_{i}^{N_{itfsSrv}} N_{methodes}^{i} \right) \\
sizeof(\texttt{ComponentIdentity}) &= (N_{itfsSrv} * sizeof(\texttt{void *})) \\
&+ sizeof(methodesCI) \\
&+ \sum_{i}^{N_{itfsSrv}} sizeof(nameof(is_i))
\end{aligned}
\tag{6.2}
$$

Où :

- $C(is)$: coût associé aux interfaces serveur d'un composant, en octets.
- $N_{itfsSrv}$: nombre d'interfaces serveur fonctionnelles du composant.
- $N_{methodes}^{is_i}$: nombre de méthodes de l'interface serveur i.
- $sizeof(methode_{ij})$: occupation mémoire du code et des données propres à l'implémentation de la méthode j de l'interface i.
- $sizeof(\texttt{void *})$: occupation mémoire d'un pointeur vers une fonction. Sur notre plate-forme de référence à 32 bits, elle est de 4 octets.
- $sizeof(methodesCI)$: occupation mémoire des implémentations des méthodes de l'interface `ComponentIdentity`. Sa valeur est indépendante du nombre d'interfaces serveur du composant.
- $sizeof(nameof(is_i))$: place mémoire nécessaire pour stocker le nom donné à l'interface serveur is_i.

Exemple. Pour le composant `sound_driver` de la figure 6.4, qui expose deux interfaces serveur `sound` et `lcc`, ainsi qu'une interface `ComponentIdentity`, en utilisant l'implémentation par défaut fournie par Fractal/Think, et compilé par GCC avec un niveau d'optimisation de `-O2`, le coût associé à la réification de ses interfaces serveur sur la plate-forme de référence est de :

$$sizeof(\texttt{srvItfsDesc}) = 4 * ((2+1)+(2+1)) = 24 \text{ octets}$$
$$sizeof(\texttt{ComponentIdentity}) = (3*4)+(38+8+8+4)+(6+4+19) = 99 \text{ octets}$$
$$\begin{aligned} C(is) &= 24 + 99 + \sum_{i}^{N_{itfsSrv}} \sum_{j}^{N_{methodes}^{is_i}} sizeof(methode_{ij}) \\ &= 123 + \sum_{i}^{N_{itfsSrv}} \sum_{j}^{N_{methodes}^{is_i}} sizeof(methode_{ij}) \\ &= 123 + sumof(sizeof(methode_{ij})) \text{ octets} \end{aligned}$$

6.3.3 Optimisations proposées

Dans la même idée que celle définie pour les attributs, nous définissons 4 niveaux d'optimisation pour la réification d'une interface serveur d'un composant. Ils sont déduits à partir de la définition des variables suivantes :

- *DESC_SRV_ITF* : le membre de la structure `srvItfsDesc` correspondant à l'interface en question est généré.
- *METHS_ITF* : l'implémentation des méthodes de l'interface en question sont ajoutées au contenu du composant
- *CI* : les méthodes de l'interface `ComponentIdentity` sont ajoutées au contenu du composant.

Les tableaux 6.4 et 6.5 présentent les 4 niveaux d'optimisation en fonction des valeurs de ces variables et de leurs descriptions. Pour la définition de ces niveaux, nous avons pris en compte les dépendances existantes entre les différentes composantes de la réification, et notamment entre l'implémentation de l'interface `ComponentIdentity` et la génération de la structure `srvItfsDesc`.

DESC_SRV_ITF	*METHS_ITF*	*CI*	**Niveau**
vrai	vrai	vrai	IS0
vrai	vrai	faux	IS1
faux	vrai	faux	IS2
faux	faux	faux	IS3

TABLE 6.4 – Variables définies et niveaux d'optimisation des interfaces serveur

Exemple. Le tableau 6.6 présente le coût associé aux réifications des interfaces serveur pour l'exemple au chapitre précèdent. Les niveaux d'optimisation sont appliqués à toutes les interfaces serveur du composant.

Considérations sur le compilateur de bas-niveau. Il est possible d'obtenir les optimisations du niveau IS3 indépendamment de la chaîne de compilation

Niveau	Description
IS0	Pas d'optimisation. Il correspond au niveau par défaut décrit au chapitre 6.3.1.
IS1	Les méthodes de l'interface `ComponentIdentity` ne sont pas ajoutées au contenu du composant. Le composant n'offre aucun service d'introspection par rapport à ces interfaces serveur. Les descripteurs d'interface sont générées.
IS2	Les méthodes de l'interface `ComponentIdentity` ne sont pas ajoutées au contenu du composant et les descripteurs d'interface ne sont pas générées. Les méthodes de l'interface serveur restent accessibles à l'intérieur du composant (e.g. par les implémentations des autres interfaces serveur ou par des méthodes privées).
IS3	L'interface serveur "disparaît", les définitions des méthodes de l'interface ne sont pas ajoutées à la réification du composant. Les descripteurs d'interface ne sont pas générées.

Table 6.5 – Description niveaux d'optimisation des interfaces serveur

Niveau	Coût de réification (octets)
IS0	$C(is) = 123 + sumof(sizeof(methode_{ij}))$ octets
IS1	$C(is) = 24 + sumof(sizeof(methode_{ij}))$ octets
IS2	$C(is) = sumof(sizeof(methode_{ij}))$ octets
IS3	$C(is) = 0$ octets

Table 6.6 – Coût de réification (en octets) des interfaces serveur de l'exemple au chapitre 6.3.2 en fonction du niveau d'optimisation appliqué.

Fractal/Think, en s'appuyant exclusivement sur le compilateur de bas-niveau. Dans GCC, cela demande des actions aux étapes ultérieures du cycle de vie, concrètement lors de la résolution de liens du code C.

A la compilation, les implémentations des composants doivent être compilées avec les options `-ffunction-sections` et `-fdata-sections`. Cela produit des sections de code et de données par fonction ou variable définis, respectivement. Ensuite, le linkeur GNU (`ld`) doit être appelé avec l'option `-gc-sections`, qui ordonne au linkeur de ne pas inclure dans le binaire final les sections non liées. Les méthodes non appelées et les données non référencées ne sont donc pas incluses dans le binaire final.

6.4 Interfaces clientes

6.4.1 Réifications par défaut

A la conception, un composant explicite le fait de requérir une interface via le mot clé `requires` (1, dans la figure 6.5). Le type de l'interface est défini dans un fichier IDL (2). Dans le contenu du composant, les méthodes des interfaces sont invoquées comme dans un appel à procédure classique (3). La correspondance entre le nom de la méthode invoquée et le nom de la méthode de l'interface est établie via une annotation sous forme de commentaire (4).

La chaîne de compilation effectue les actions suivantes pour réifier les interfaces clientes :

i. Une structure de données qui joue le rôle de descripteur des interfaces clientes, `cltItfsDesc`, est générée dynamiquement par la chaîne de compilation (5). Chaque membre de cette structure fait référence à une interface cliente du composant. Il est initialisé avec l'adresse du descripteur de l'interface serveur à laquelle l'interface cliente est liée, i.e. l'adresse du membre adéquat d'une structure du type `srvItfsDesc` (cf. 6.5).

ii. Les invocations aux méthodes des interfaces clientes sont transformées par la chaîne de compilation. Elles sont désormais faites par indirections successives via la structure de données précédemment décrite (6). Ce processus est décrit au chapitre 6.5.

6.4.2 Coût associé aux réifications

Nous évaluons ici le coût associé aux interfaces clientes en termes d'occupation mémoire de la structure de description `cltItfsDesc`. Le coût associé à l'invocation des méthodes clientes est étudié au chapitre 6.5.

$$\begin{aligned} C(ic) &= sizeof(\texttt{cltItfsDesc}) \\ &= N_{itfsClt} * sizeof(\texttt{void *}) \end{aligned} \tag{6.3}$$

Où :

- $C(ic)$: coût associé aux interfaces clientes d'un composant, en octets.
- $N_{itfsClt}$: nombre d'interfaces clientes fonctionnelles du composant.
- $sizeof(\texttt{void *})$: occupation mémoire d'un pointeur vers une fonction. Sur notre plate-forme de reference à 32 bits, elle est de 4 octets.

Exemple. Le composant `sound_driver` de la figure 6.4 au chapitre précèdent requiert trois interfaces clientes `irq`, `lock` et `delay`. Compilé par GCC avec un niveau d'optimisation de `-O2`, le coût associé à la réification de ses interfaces clientes sur notre plate-forme matérielle de référence est de :

$$C(ic) = 3 * 4 = 12 \text{ octets}$$

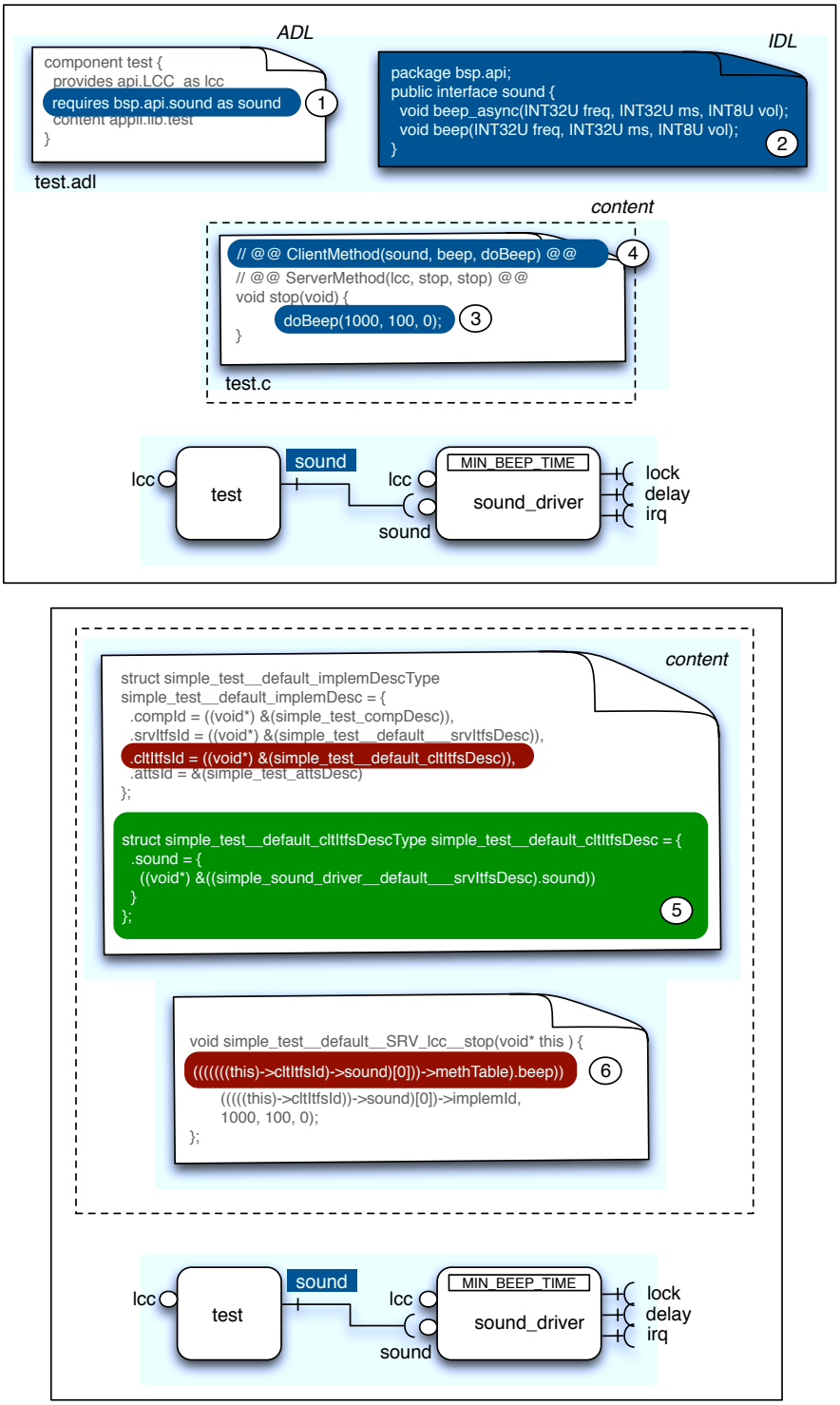


FIGURE 6.5 – Extraits des réifications du composant `test` à la conception (haut) et après la première phase de compilation (bas). Les éléments liés à l'interface cliente `sound` sont mis en relief.

6.4.3 Optimisations proposées

Même si les interfaces clientes restent beaucoup plus modestes que les interfaces serveurs, elles peuvent être optimisées. Les descripteurs d'interfaces clientes générées par la chaîne de compilation sont nécessaires à la construction des liens entre composants. L'optimisation des interfaces clientes est donc étroitement liée à l'optimisation des liaisons. C'est l'objet du chapitre suivant.

6.5 Liaisons

6.5.1 Réifications par défaut

A la conception, les liaisons entre deux composants par des interfaces du même type, sont définies dans la description du composite qui contient les deux composants liés (1, dans la figure 6.6). Dans Fractal/Think les liaisons sont dirigées du composant client vers le composant serveur (2).

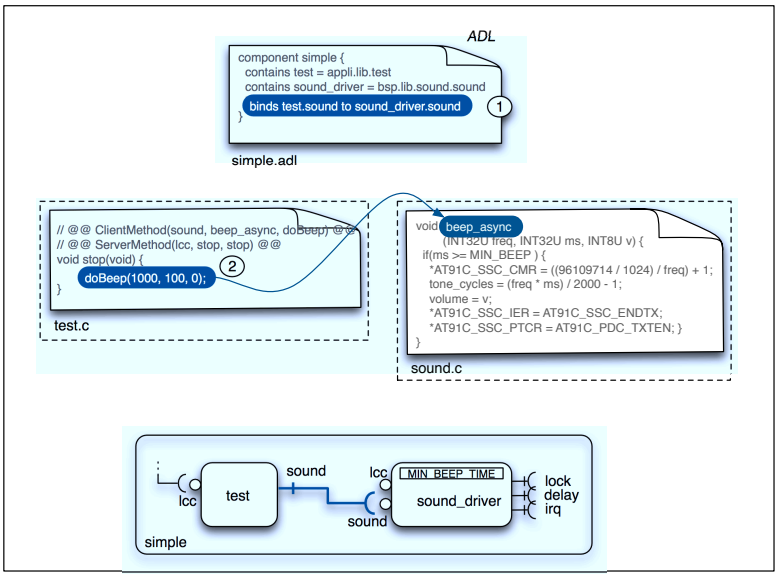
La chaîne de compilation effectue les actions suivantes pour réifier les liaisons :

i. Dans le composant client, l'invocation des méthodes de l'interface cliente est construite à partir des descripteurs des interfaces (cf. 6.3.1 et 6.4.1) :

   1. Le descripteur de l'interface cliente pour qui la méthode invoquée est associée, est récupéré à partir du descripteur de l'instance du composant (3.1). Cette action est décrite au chapitre 6.7.
   2. L'adresse du *descripteur* de l'interface serveur à qui l'interface cliente est liée, est récupérée (3.2). Le type de cette adresse (`void *`) est converti à celui du descripteur de l'interface serveur.
   3. L'adresse de l'*implémentation* de la méthode serveur recherchée est obtenue à partir de la table de méthodes du descripteur de l'interface serveur. Cette méthode est invoquée via un pointeur sur fonction (3.3).

ii. Des méthodes permettant la modification de la direction du lien et d'obtenir des informations relatives aux liaisons, sont ajoutées au contenu du composant client. Ces méthodes sont regroupées dans une interface de type `BindingController` :

```
public interface BindingController {
  // bind component client interface identified by
  // clientItfName to a server interface serverItfId
  void bind(ADL_ID_TYPE clientItfName, any serverItfId);
  // unbind
  void unbind(ADL_ID_TYPE clientItfName);
  // get identifier of server interface bound to
  // client interface identified by clientItfName
  any lookup(ADL_ID_TYPE clientItfName);
  // get a list of client interfaces names
  any* list(void);
  // get number of client interfaces
  int size(void);
}
```



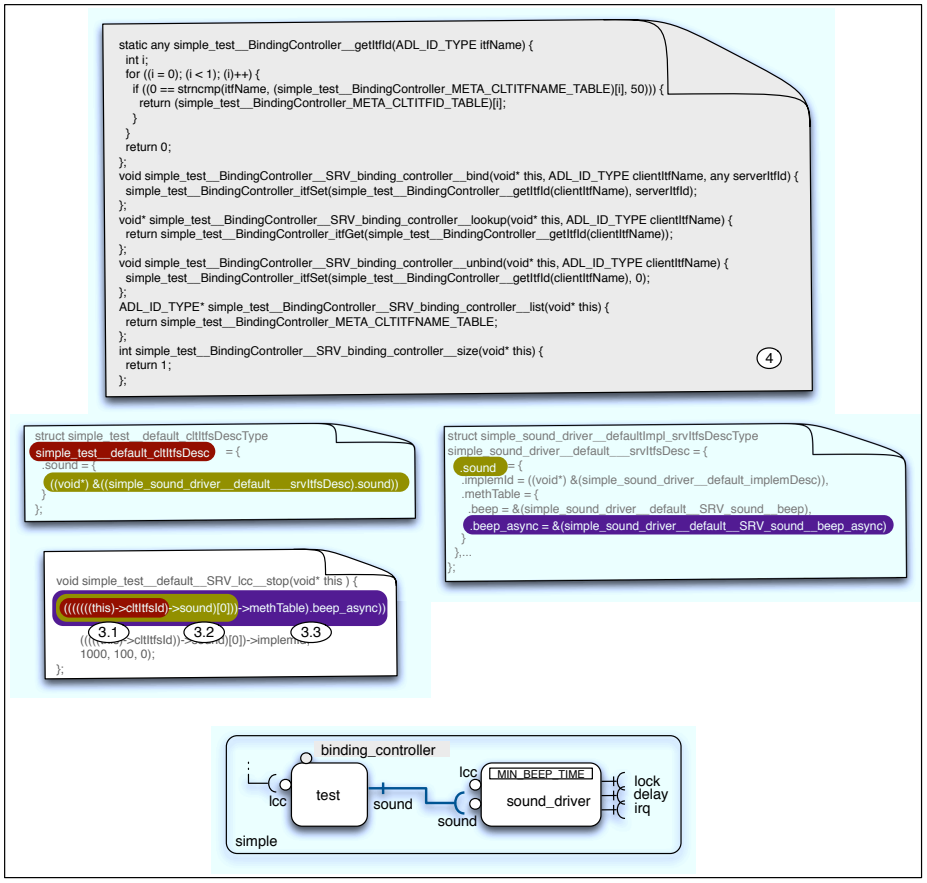


FIGURE 6.6 – Extraits des réifications des composants primitifs `sound_driver` et `test`, à la conception (haut) et après la première phase de compilation (bas). Les éléments liés à la liaison entre ces deux composants sont mis en relief.

En l'absence de code fourni par les développeurs, la chaîne de compilation génère des implémentations par défaut de ces méthodes.

6.5.2 Coût associé aux réifications

Nous calculons le coût associé à l'ensemble de liaisons dirigées à partir d'un composant donné. Ce coût est estimé par rapport à deux critères. D'un coté, l'occupation mémoire due à leurs réifications, dont la valeur dépend d'une partie des coûts liés aux interfaces clientes et serveur qui sont liées, et du coût lié à l'interface `BindingController`. Ce coût $C(l_m em)$ est donné par :

$$\begin{aligned} C(l_m em) &= sizeof(\texttt{cltItfsDesc}) \\ &+ \sum_{i}^{N_{liaisons}} sizeof(srvItfDesc_i) \\ &+ sizeof(\texttt{BindingController}) \\ sizeof(\texttt{BindingController}) &= sizeof(methodesBC) \\ &+ sizeof(methodesBC_P RV) \\ &+ \sum_{i}^{N_{liaisons}} sizeof(nameof(ic_i)) \\ &+ (N_{liaisons} * sizeof(\texttt{void *})) \end{aligned} \tag{6.4}$$

Où :

- $C(l_m em)$: coût associé à l'occupation mémoire des liaisons clientes d'un composant, en octets.
- $sizeof(\texttt{cltItfsDesc})$: coût associé au descripteur des interfaces clientes du composant [3].
- $N_{liaisons}$: nombre des liaisons qui partent du composant.
- $nameof(ic_i)$: chaîne de caractères du nom donné à l'interface cliente concernée par la liaison i du composant.
- $sizeof(srvItfDesc_i)$: coût relatif dans la structure `srvItfDesc` du composant serveur de l'interface vers qui la liaison i se dirige.
- $sizeof(methodesBC)$: occupation mémoire due aux implémentations des méthodes de l'interface `BindingController`. Sa valeur est indépendante du nombre de liaisons où le composant client de la liaison en question joue le rôle de composant client.
- $sizeof(methodesBC_P RV)$: occupation mémoire due aux implémentations des méthodes privées nécessaires aux méthodes de l'interface `BindingController`.
- $sizeof(\texttt{void *})$: occupation mémoire d'un pointeur vers une fonction. Sur notre plate-forme de référence à 32 bits, elle est de 4 octets.

3. Si certaines interfaces clientes du composants ne sont liées à aucune interface serveur, leur coût relatif est déduit de cette valeur.

D'un autre coté, le nombre d'instructions nécessaires pour effectuer l'invocation d'une méthode d'une interface augmente (par rapport à un appel direct de fonction), du fait de devoir récupérer l'adresse de la méthode serveur adéquate via des structures de méta-données générées par la chaîne de compilation. Cela produit une augmentation de l'occupation mémoire du contenu du composant qui se reflète dans le coût associé soit aux interfaces serveur (cf. 6.3.1), soit aux méthodes privées.

Cela affecte également le temps d'exécution de la méthode qui invoque la méthode cliente, en retardant le moment où la méthode serveur appropriée est effectivement exécutée. Le temps d'exécution additionnel dépendra de la manière dont le code C est compilé en code binaire, des optimisations appliquées par la chaîne de compilation de bas-niveau et des caractéristiques de la plate-forme matérielle sous-jacente (pipeline d'instructions, fréquence d'horloge, jeux d'instructions utilisé...).

Exemple. Le composant `test` de la figure 6.6 n'est lié qu'à un seul composant, `sound_driver`, via une interface de type `sound`. En utilisant l'implémentation par défaut de l'interface `BindingController` fournie par Fractal/Think, et compilé par GCC avec un niveau d'optimisation `-O2`, le coût associé à la réification de ses liaisons sur notre plate-forme matérielle de référence, est donné par :

$$
\begin{aligned}
sizeof(\texttt{cltItfsDesc}) &= 1 * 4 = 4 \text{ octets} \\
\sum_{i}^{N_{liaisons}} sizeof(srvItfDesc_i) &= (3 * 4) = 12 \text{ octets} \\
sizeof(\texttt{BindingController}) &= (14 + 10 + 10 + 8 + 4) + (24) + (24) + (1 * 4) \\
&= 98 \text{ octets} \\
C(l_m em) &= 4 + 12 + 98 = 114 \text{ octets}
\end{aligned}
$$

Le code suivant présente le code assembleur de la méthode `start` de l'interface `lcc` du composant `test`, qui invoque simplement la méthode `beep` de son interface cliente de type `sound`, liée au composant `sound_driver`. On compte 11 instructions devant être exécutées. Au chapitre suivant ce code sera comparé avec le code obtenu après optimisation.

```
002035d8 <simple_test__default__SRV_lcc__start>:
 2035d8:  e5903008  ldr r3, [r0, #8]
 2035dc:  e590200c  ldr r2, [r0, #12]
 2035e0:  e593c000  ldr ip, [r3]
 2035e4:  e5921004  ldr r1, [r2, #4]
 2035e8:  e5d23008  ldrb r3, [r2, #8]
 2035ec:  e52de004  push {lr} ; (str lr, [sp, #-4]!)
 2035f0:  e59c0000  ldr r0, [ip]
 2035f4:  e5922000  ldr r2, [r2]
 2035f8:  e1a0e00f  mov lr, pc
 2035fc:  e59cf004  ldr pc, [ip, #4]
 203600:  e49df004  pop {pc} ; (ldr pc, [sp], #4)
```

6.5.3 Optimisations proposées

Nous définissons 6 niveaux d'optimisation pour la réification d'une liaison. Ils sont déduits à partir de la définition des variables suivantes :

- *DESC_SRV_ITF* : le membre de la structure `srvItfsDesc` correspondant à l'interface serveur de la liaison en question est généré et ajouté au contenu du composant serveur.
- *DESC_CLT_ITF* : le membre de la structure `cltItfsDesc` correspondant à l'interface cliente de la liaison en question, est généré et ajouté au contenu du composant client.
- *BC_CONSU* : les méthodes de consultation de l'interface `BindingController` (`lookup`, `list`, `size`) sont ajoutées au contenu du composant client.
- *BC_MODIF* : les méthodes de modification de l'interface `BindingController` (`bind` et `unbind`) sont ajoutées au contenu du composant client.

Les tableaux 6.7 et 6.8 présentent les valeurs de ces variables qui définissent les six niveaux d'optimisation et une description des niveaux, respectivement. Dans la définition de ces niveaux nous avons pris en compte les dépendances existantes entre les différentes composantes de la réification, et notamment entre l'implémentation de l'interface `BindingController` et la génération de la structure `cltItfsDesc`.

DESC_SRV_ITF	*DESC_CLT_ITF*	*BC_CONSU*	*BC_MODIF*	**Niveau**
vrai	vrai	vrai	vrai	L0
vrai	vrai	vrai	faux	L1
vrai	vrai	faux	vrai	L2
vrai	vrai	faux	faux	L3
x	faux	faux	faux	L4
x	faux	faux	faux	L5

TABLE 6.7 – Variables définies et niveaux d'optimisation des liaisons

Exemple. Le tableau 6.9 présente le coût associé à la réification de la liaison partant du composant `test` de la figure 6.6.

Concernant le deuxième critère de coût de réification, i.e. le temps d'exécution, le code suivant présente le code assembleur de la méthode `start` de l'interface `lcc` du composant `test`. Cette fois la liaison a été compilée au niveau d'optimisation L4. On constate que le nombre d'instructions à être exécutées diminue de 11 à 8.

Niveau	Description
L0	Pas d'optimisation. Correspond au niveau par défaut décrit au chapitre 6.5.1.
L1	La direction du lien n'est pas modifiable, mais elle reste consultable.
L2	La direction du lien n'est pas consultable, mais elle reste modifiable.
L3	Le lien n'est ni consultable ni modifiable. Les méta-données sur les interfaces clientes et serveur sont générées et l'invocation des méthodes clientes est faite via ces structures de données.
L4	Comme le niveau L3, mais les méta-données sur l'interface cliente ne sont pas générées. Indépendamment du fait que les méta-données sur l'interface serveur soient générées ou non, l'invocation des méthodes de l'interface cliente est réifiée comme un simple appel à la méthode appropriée du composant serveur.
L5	L'implémentation de la méthode serveur est embarquée dans le contenu du composant client. Cela est fait via la génération de versions *inline* des méthodes de l'interface serveur, en ajoutant à leur déclaration le mot clé ANSI C `inline`. Le compilateur de bas-niveau essaie d'intégrer les instructions de la méthode serveur au contenu du composant client. On peut dire donc que le lien « disparaît » à la compilation. Le choix d'implémentation de cette optimisation est inspiré du fait que les outils disponibles dans la plate-forme Fractal/Think ne permettent une analyse exhaustive du comportement des composants, nécessaire afin d'intégrer de manière sûre une fonction dans la définition d'une autre.

Table 6.8 – Description niveaux d'optimisation des liaisons

Niveau	Coût de réification (octets)
L0	$C(l_m em) = 4 + 12 + 98 = 114$
L1	$C(l_m em) = 4 + 12 + ((10 + 8 + 4) + 24 + 24 + 4) = 90$
L2	$C(l_m em) = 4 + 12 + ((14 + 10) + 24 + 24 + 4) = 92$
L3	$C(l_m em) = 4 + 12 = 16$
L4	$C(l_m em) = 0$ ou 16
L5	$C(l_m em) = 0$ ou 16

Table 6.9 – Coût de réification de la liaison de l'exemple au chapitre 6.5.2 en fonction du niveaux d'optimisation appliqué.

```
00203600 <simple_test__default__SRV_lcc__start>:
 203600:  e5903008  ldr r3, [r0, #8]
 203604:  e590200c  ldr r2, [r0, #12]
 203608:  e5933000  ldr r3, [r3]
 20360c:  e5921004  ldr r1, [r2, #4]
 203610:  e5930000  ldr r0, [r3]
 203614:  e5923008  ldr r3, [r2, #8]
 203618:  e5922000  ldr r2, [r2]
 20361c:  ea0006c2  b 20512c <simple_sound_driver
                              __default__SRV_sound__beep>
```

Considérations sur le compilateur de bas-niveau. L'obtention d'un niveau d'optimisation L5 dépend fortement du comportement des compilateurs de bas-niveau. Dans GCC, le niveau d'optimisation `-O2` active les options de compilation `-finline-small-functions` et `-findirect-inlining` ; le niveau `-O3` active, en plus de celles-ci, l'option `-finline-functions`. Dans les deux cas, marquer les méthodes serveur comme *inline* ne garantit pas que le compilateur effectuera cette optimisation. En effet, il effectue des analyses heuristiques pour prendre la décision d'optimiser dont nous ne disposons pas d'informations.

6.6 Composites

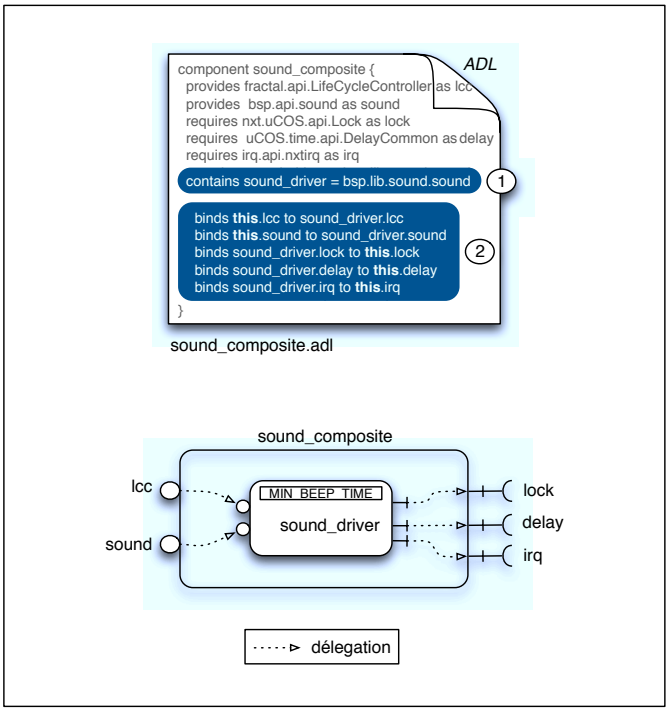
6.6.1 Réifications par défaut

Le modèle Fractal/Think supporte l'assemblage de composants en forme de composites. Cette encapsulation de composants est définie à la conception, dans le fichier ADL du composite, via l'inclusion d'un sous-composant dans le contenu du composite via le mot clé `contains` (1, dans la figure 6.7) et la délégation des services offerts/requis par le composite, via des liaisons à partir de/vers un composant virtuel représentant le composite, identifié par le mot clé `this` (2) ; ces services sont implémentés ou invoqués par les sous-composants du composite.

Afin de réifier la délégation de services et le composant virtuel `this`, la chaîne de compilation Fractal/Think effectue les actions suivantes :

i. Pour chaque interface serveur du composite qui est déléguée vers un sous-composant :

   1. Une interface cliente du même type, appelée « interface interne », est créée. Le nom donné à cette interface est celui de l'interface serveur préfixé par `internal_` (3.1).
   2. L'implémentation de l'interface serveur du composite est générée. Pour chaque méthode de l'interface, le comportement consiste à faire appel à la méthode correspondante de l'interface cliente (3.2).
   3. Une liaison est créée entre l'interface interne et l'interface serveur du sous-composant vers qui est délégué le service (3.3). Ainsi, une fois cette liaison réifiée (cf. 6.5), l'implémentation de l'interface serveur du composite fait appel au sous-composant approprié.



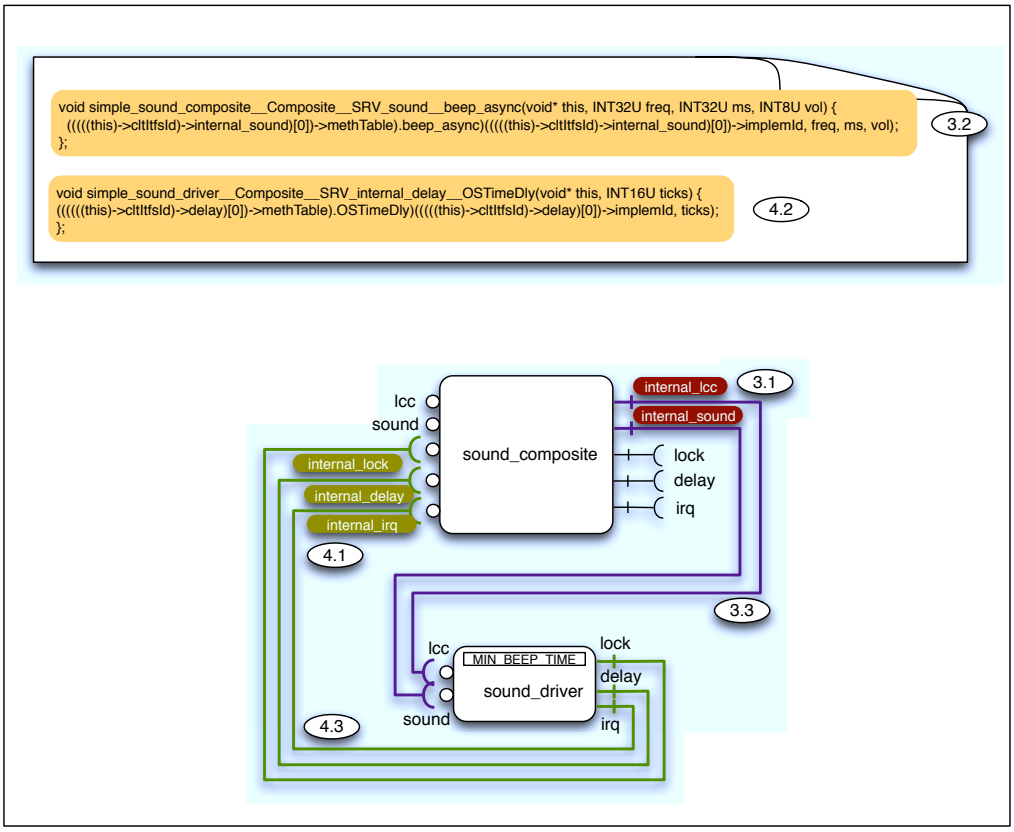


FIGURE 6.7 – Extraits des réifications du composite `sound_composite` à la conception (haut) et après la première phase de compilation (bas).

ii. Pour chaque interface cliente du composite, une procédure similaire est effectuée :

1. Une interface serveur interne du même type est crée. Le nom donné à cette interface est celui de l'interface serveur préfixé par `internal_` (4.1).
2. L'implémentation de cette interface serveur interne du composite est générée. Pour chaque méthode de l'interface, le comportement consiste à faire appel à la méthode correspondante de l'interface cliente (4.2).
3. Une liaison entre l'interface cliente du sous-composant et cette interface serveur interne est créée (4.3).

Les liaisons et les interfaces créés, ainsi que les interfaces qui ne sont pas délégues, sont ensuite réifiées comme décrit aux chapitres 6.4, 6.3 et 6.5.

La chaîne de compilation Fractal/Think effectue également l'action suivante concernant les composites :

iii. Des méthodes permettant la modification de l'assemblage interne du composite (`addSubComponent`, `removeSubComponent`) et l'obtention d'informations sur celui-ci (`getSubComponent`, `list`, `size`) sont ajoutées au contenu du composite. Ces méthodes sont regroupées dans une interface de type `ContentController`. En l'absence de code fourni par les développeurs, la chaîne de compilation génère des implémentations par défaut.

```
public interface ContentController {
  // add a component into a composite
  void addSubComponent(ADL_ID_TYPE compName, any subCompId);
  // remove a component from a composite
  any    removeSubComponent(ADL_ID_TYPE compName);
  // get a subcomponent id from its name
  any    getSubComponent(ADL_ID_TYPE compName);
  // list subcomponents names
  ADL_ID_TYPE* list(void);
  // get number of subcomponents
  int size(void);
}
```

L'implémentation des méthodes de l'interface `ContentController` demande la génération d'un ensemble de structures de méta-données concernant les sous composants. En particulier, l'implémentation par défaut génère un tableau `ContentController_META_SUBCOMPID_TABLE`, dont les membres sont initialisés avec les descripteurs des interfaces serveur `ComponentIdentity` des sous-composants.

Également, une fois un sous-composant a été ajouté (ou enlevé) de l'assemblage interne du composite, les liaisons partant des interfaces clientes internes doivent être modifiées afin de rendre ce sous-composant (non) opérationnel. Cela oblige au composite d'ajouter à son contenu les méthodes de l'interface `BindingController`.

6.6.2 Coût associé aux réifications

Le coût associé à la réification d'un composite est donné par :

$$
\begin{aligned}
C(co) &= C(is) \\
&+ C(ic) \\
&+ sizeof(\texttt{BindingController}) \\
&+ sizeof(ContentController) \\
sizeof(ContentController) &= sizeof(methodesCC) \\
&+ sizeof(methodesCC_PRV) \\
&+ \sum_{i}^{N_{souscomps}} sizeof(nameof(sc_i)) \\
&+ (N_{souscomps} * sizeof(\texttt{void *}))
\end{aligned}
\tag{6.5}
$$

Où :

- $C(co)$: coût associé à la réification du composite, en octets.
- $C(is)$: coût associé à la réification des interfaces serveur du composite (internes incluses), calculé comme dans 6.3.2.
- $C(is)$: coût associé à la réification des interfaces clientes du composite (internes incluses), calculé comme dans 6.4.2.
- $sizeof(\texttt{BindingController})$: occupation mémoire de l'interface `BindingController` qui doit être ajoutée au contenu du composite. Elle est calculée comme dans 6.5.2.
- $sizeof(methodesCC)$: occupation mémoire due aux implémentations des méthodes de l'interface `ContentController`.
- $sizeof(methodesCC_PRV)$: occupation mémoire due aux implémentations des méthodes privées nécessaires aux méthodes de l'interface `ContentController`.
- $N_{souscomps}$: nombre de sous-composants du composite.
- $nameof(sc_i)$: chaîne de caractères du nom du sous-composant i du composite.
- $sizeof(\texttt{void *})$: occupation mémoire d'un pointeur le descripteur de l'interface `ComponentIdentity` d'un sous-composant. Cette valeur dépend de la place mémoire donnée à un pointeur dans la plate-forme matérielle. Sur 32 bits, elle est de 4 octets.

A noter que le coût des liaisons générées à la compilation n'est pas inclus explicitement dans le calcul. En effet, il est considéré implicitement dans les calculs de $C(is)$, $C(ic)$ et $sizeof(\texttt{BindingController})$.

Exemple. Le composite `sound_composite` de la figure 6.7 contient un seul sous-composant `sound_driver`. En utilisant l'implémentation par défaut de l'interface `ContentController` fournie par Fractal/Think et compilé par GCC avec un niveau d'optimisation de `-O2`, le coût associé à la réification de ce composite sur notre plate-forme matérielle de référence est de :

$$
\begin{aligned}
sizeof(\texttt{ContentController}) &= (28+4+44+8+12)+0+13+(1*4) = 113 \text{ octets} \\
C(is) &= ((7*4)+(3*4)+(4*4)+(3*4)+(5*4)) \\
&\quad + ((5*4)+(38+8+8+4)+52)+(772) \\
&= 990 \text{ octets} \\
C(ic) &= (5*4) = 20 \\
sizeof(\texttt{BindingController}) &= (14+10+10+8+4)+(24)+(43)+(5*4)) \\
&= 133 \text{ octets} \\
C(co) &= 113+990+20+133 = 1256 \text{ octets}
\end{aligned}
$$

6.6.3 Optimisations proposées

Nous définissons trois niveaux d'optimisation pour les composites en fonction de deux variables. D'abord, l'ajout ou pas de l'interface `ContentController` dans le contenu du composite. Ensuite, dans le cas ou cette interface n'y est pas ajoutée, nous définissons deux sous-niveaux en fonction de la génération ou non des méthodes de délégation de services. Le tableau 6.11 décrit les niveaux définis.

Exemple. Le tableau 6.10 présente le coût associé à la réification du composite `sound_composite` de la figure 6.7 lorsqu'on applique les niveaux d'optimisation définis.

Niveau	**Coût de réification (octets)**
CO0	$C(co) = 1256$
CO1.1	$C(co) = 1256 - 133 = 1123$
CO1.2	$C(co) = 0$

Table 6.10 – Coût de réification du composite de l'exemple au chapitre 6.6.2, en fonction du niveaux d'optimisation appliqué.

Considérations sur le compilateur de bas-niveau.. Dans la mesure où la réification des composants composite implique la création des liaisons et d'interfaces serveur et clientes, les considérations sur les compilateurs de bas-niveau faites aux chapitres précédents sont également valides pour les composites.

Niveau	Description
CO0	Pas d'optimisation. Correspond au niveau par défaut décrit au chapitre 6.6.1.
CO1	les méthodes de l'interface `ContentController` ne sont pas ajoutées au contenu du composite. Également, les méta-données sur les sous-composants ne sont pas générées. Ce niveau d'optimisation offre néanmoins un bon nombre de variations en fonction des niveaux d'optimisation des interfaces serveur internes générées dynamiquement (cf. ii.1 et ii.2 au chapitre 6.6.1) et celui des liaisons, soit entre les interfaces clientes internes et les sous-composants (cf. i.3), soit entre ceux-ci et les interfaces serveur internes (cf. ii.3). Le niveau d'optimisation des liaisons où le composite joue le rôle de client varie également le niveau d'optimisation du composite. Nous définissons deux sous-niveaux CO1.1 et CO1.2, en fonction de la création ou non des méthodes de délégation.
CO1.1	Les implémentations des méthodes de délégation (cf. i.2 et ii.2) sont générées comme décrit précédemment. Aux interfaces serveur et aux liaisons générées à la compilation on applique des niveaux d'optimisation de IS1 et L3, respectivement. Nous supposons que les liaisons où le composite est le composant client ont elles aussi un niveau d'optimisation L3 ou supérieur, ce qui implique que l'interface `BindingController` n'est pas ajoutée au contenu du composite.
C01.2	Ce niveau implique la « disparition » des délégations de services, dans la mesure où les composants demandant les services du composite sont directement liés au sous-composant qui implémente le service. De façon analogue, les sous-composants sont directement liés aux composants qui implémentent les services que ceux-ci demandent. Dans le cas ou le composite n'a pas de contenu autre que ses sous-composants, le composite disparaît.

Table 6.11 – Description niveaux d'optimisation des composites

6.7 Partage de code entre instances de composants

6.7.1 Réifications par défaut

Les composants Fractal/Think sont *multi-instanciables* par défaut. A la conception, le mot clé `contains` (1, dans la figure 6.8) définit l'inclusion d'un composant dans l'assemblage d'un composite ; dans un sens plus strict, une *instance* d'un *type* de composant est ajoutée au contenu du composite, et un nom unique dans le contexte du composite est assignée à celle-ci (2). Dans la figure 6.8 deux instances du type de composant `bsp.lib.sound.sound` sont ajoutées, elles sont nommées `sound_driver1` et `sound_driver2`.

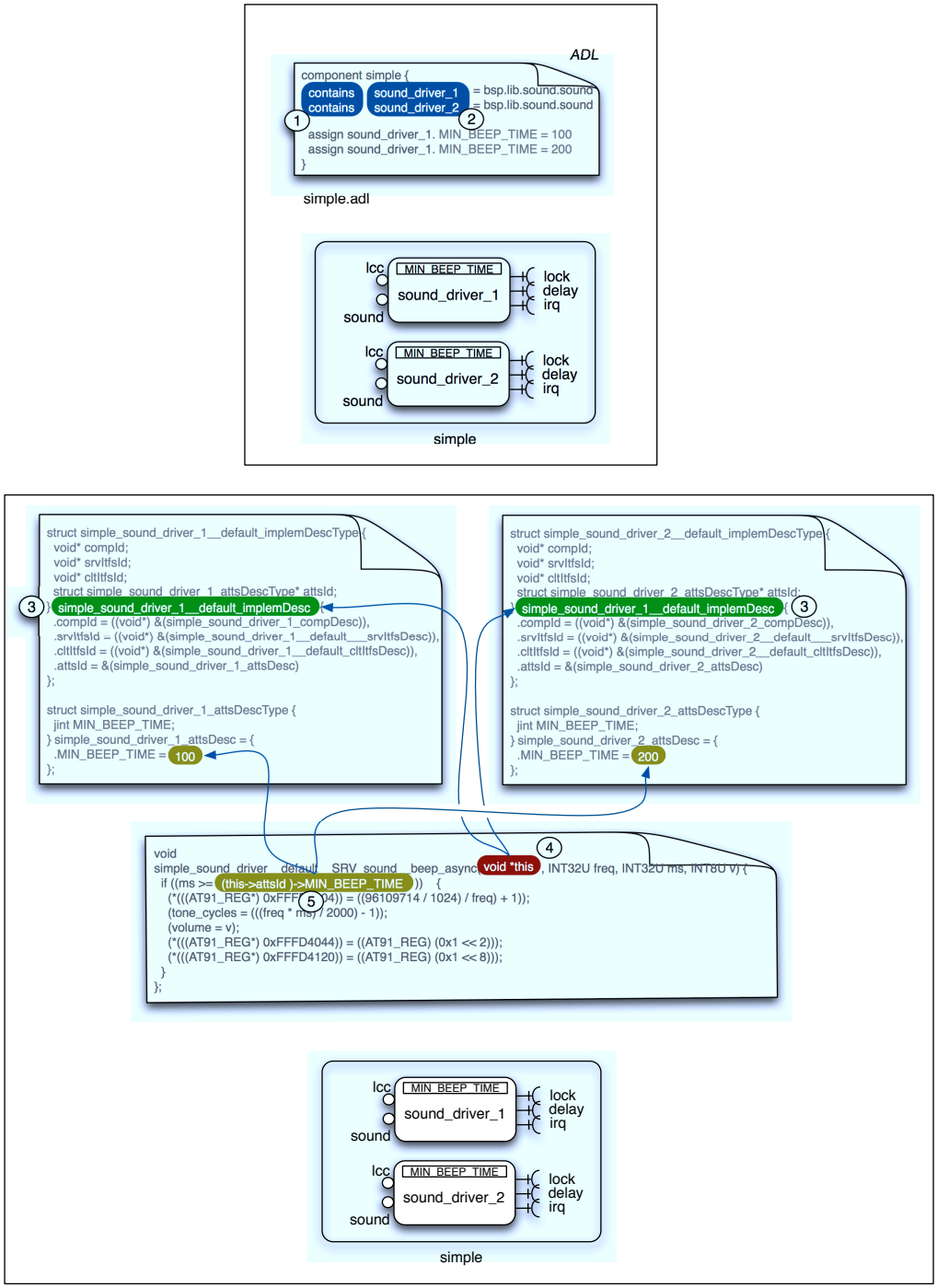


FIGURE 6.8 – Réification de deux instances du même type de composant, ici `sound_driver1` et `sound_driver2`. Les flèches montrent les valeurs de la variable `this`, ainsi que les indirections pour accéder à la valeur d'un attribut, lorsque le code partagé est exécuté dans le contexte des deux composants.

Cette notion d'instance dans le modèle Fractal/Think se matérialise par le partage des implémentations, i.e. du code correspondant aux méthodes des interfaces serveur et aux méthodes privées, entre les composants du même type. Ce code n'est généré qu'une seule fois par la chaîne de compilation ; on parle donc de partage de code. Par contre, les méta-données sur les entités des composants étudiées aux chapitres précédents (`attsDesc`, `srvItfsDesc`, `cltItfsDesc`) sont générées pour chaque instance du type de composant incluse dans l'application [4].

Par la suite, et comme cela a été le cas au long de cette thèse, nous parlerons de « composant » pour designer une instance d'un type de composant, et du « type du composant » quand la référence au type duquel se dérive le composant sera nécessaire.

La chaîne de compilation réifie le partage de code entre instances de composants en effectuant les actions suivantes :

i. Une structure de méta-données `implemDesc` est générée pour chaque composant (3). Elle est initialisée avec des pointeurs vers les structures de méta-données du composant (`attsDesc`, `srvItfsDesc`, `cltItfsDesc`).

ii. Les déclarations des méthodes d'interfaces serveur et des méthodes privées du composant sont modifiées. Un paramètre d'entrée appelé `this`, de type `void *`, est ajouté comme premier argument d'entrée aux méthodes (4). Cette variable `this` est en fait un pointeur vers la structure `implemDesc` du composant, elle joue le rôle de descripteur du composant.

iii. Les réifications des références aux attributs (cf. 6.2) et des invocations des méthodes clientes (cf. 6.4 et 6.5) se voient affectées par le partage de code entre composants :

   (a) Les références aux attributs se font à travers de la variable `this`. A partir de celle-ci on récupère le descripteur des attributs `attsDesc`, dont l'adresse est stockée au champ `attsId` de `implemDesc`. A partir du descripteur d'attributs on accède à l'attribut en question de chaque composant, comme décrit au chapitre 6.2 (5).

   (b) Les invocations aux méthodes clientes demandent deux actions concernant le partage de code :
      - Le descripteur des interfaces clientes `cltItfsDesc` est récupéré à partir du `this`, puisque son adresse est stockée au champ `cltItfsId` de `implemDesc`. A partir de ce descripteur on récupère le descripteur de l'interface en question. La construction de la liaison continue comme décrit au chapitre 6.5.
      - Le descripteur du composant serveur doit être passé à la méthode invoquée comme premier paramètre. Il est récupéré à partir du descripteur de l'interface serveur `srvItfsDesc` du composant invoqué. `srvItfsDesc` est à son tour récupéré à partir du descripteur de l'interface cliente. En effet, le membre de `srvItfsDesc` qui fait référence à l'interface ser-

4. Par défaut, le génération de code faite par Fractal/Think suppose que le code partagé est un code reentrant.

veur en question, stocke un pointeur vers la structure `implemDesc` du composant, i.e. vers le descripteur de celui-ci.

iv. Un composant de type `Factory` est ajouté pour chaque type de composant présent dans l'application. Ce composant offre une interface du type `fractal.api.Factory` qui offre des services de création de nouveaux composants de ce type. Il requiert également d'un service d'allocation de mémoire. L'implémentation de ce composant est générée dynamiquement par la chaîne de compilation.

```
public interface Factory {
    // instanciate a component, returns the descriptor
    // of its ComponentIdentity interface
    any instanciate(void);
    // instanciate a component, returns the identifier
    // of its itfName server interface
    any instanciateAndGet(ADL_ID_TYPE itfName);
    // destroy a created component
    void destroy(any compId);
    // reintialize a created component
    void reInit(any compId);
}
```

6.7.2 Coût associé aux réifications

Le partage de code entre composants du même type affecte la manière dont la chaîne de compilation réifie les entités du modèle à composants, notamment en ce qui concerne les attributs et les liaisons. Également, l'implémentation des composants `Factory` dépend fortement des caractéristiques du type de composant auquel il est dédié (nombre d'attributs, d'interfaces serveur et des méthodes de ces interfaces). C'est pourquoi il est difficile de fournir un calcul généralisable pour estimer le coût associé au partage de code dans un type de composant.

L'impact du `this` porte sur les indirections nécessaires afin d'accéder aux métadonnées sur les entités des composants, ainsi que par le paramètre additionnel à rajouter dans la déclaration de toutes les méthodes serveur et privées du composant. Comme dans le cas des indirections liées à la construction des liaisons (cf. 6.5), cela affecte l'occupation mémoire et le temps d'exécution des méthodes, puisque le nombre d'instructions nécessaires pour accomplir la tâche est supérieur. Néanmoins, le nombre exact d'instructions nécessaires dépend très fortement de la chaîne de compilation de bas-niveau et des options d'optimisation activées.

6.7.3 Optimisations proposées

Nous définissons 4 niveaux d'optimisation concernant les composants de même type. Ils sont déduits à partir des variables booléennes suivantes :

- *HAS_FACTORY* : un composant du type `Factory` associé au type du composant en question est généré.
- *THIS_USED* : le paramètre `this` est ajouté aux déclarations des méthodes privées et serveur du composant. Le `this` est utilisé pour accéder aux attributs du composant.

Les tableaux 6.12 et 6.13 présentent les valeurs de ces variables qui définissent les quatre niveaux d'optimisation et une description des niveaux, respectivement.

HAS_FACTORY	*THIS_USED*	**Niveau**
vrai	vrai	P0
vrai	faux	P1
faux	faux	P2.1 ou P2.2

TABLE 6.12 – Variables définies et niveaux d'optimisation des composants multi/-mono instance

Considérations sur le compilateur de bas-niveau.. Comme cela a été évoqué précédemment, l'impact du `this` sur la performance des composants est lié aux indirections nécessaires pour accéder aux descripteurs des entités à l'exécution. Le nombre exact d'instructions nécessaires pour réifier ces indirections est déterminé par la chaîne de compilation de bas-niveau, et en particulier par les options d'optimisation activées.

6.8 Synthèse

Nous identifions quatre aspects qui nous semblent particulièrement pertinents concernant nos études et propositions d'optimisation des réifications à la compilation au sein de la plate-forme Fractal/Think.

Flexibilité de l'optimisation. D'abord, nous constatons qu'il est possible d'obtenir, à partir de la réification d'un composant à la conception, i.e. à partir de sa description à haut niveau, plusieurs versions de ses réifications à la compilation qui mènent à des profils variés de performance du composant à l'exécution, notamment en ce qui concerne l'occupation mémoire et le temps d'exécution.

Cela est un pas en avant par rapport aux optimisations menées par les plates-formes de développement à base de composants pour l'embarqué étudiées au chapitre 3 : les niveaux d'optimisation de chacune des entités du modèle, définis aux chapitres 6.2-6.7, nous offrent une palette de techniques d'optimisation qui peuvent être déclenchées individuellement (sauf pour les cas où ils existent des dépendances entre optimisations inter-entités, cas qui ont été identifiés ici et qui ne sont pas nombreux), contrairement à l'application systématique de techniques d'optimisation dans les plate-formes de référence.

Par ailleurs, le fait d'offrir des niveaux d'optimisation au niveau des entités du modèle et non du composant ou de l'application entière, permet de régler plus finement la performance de l'application en fonction de ses besoins fonctionnels et extra-fonctionnels. Ici nous avons traité exclusivement l'aspect optimisation du logiciel, au chapitre 7 nous considérons en outre l'aspect Evolution à l'exécution.

Niveau	Description
P0	Correspond au niveau par défaut décrit au chapitre 6.7.1.
P1	Le composant de type `Factory` n'est pas généré. Cela implique que le nombre de composants d'un même type est figé à la conception et que ces composants partagent le code d'implémentation. Si l'on voulait considérer ce partage de code comme une optimisation, il serait nécessaire de comparer le gain en occupation mémoire de ne générer qu'une seule fois le code d'implémentation, par rapport au coût, en termes d'occupation mémoire, associé à l'utilisation du `this`. Dans la plupart des cas cette comparaison est équivalente à dire que le nombre de composants d'un type donné est supérieur à 1.
P2	Le paramètre `this` n'est pas ajouté aux déclarations des méthodes de l'implémentation du composant. Il n'est pas utilisé pour accéder aux attributs. Le partage de code n'est donc pas possible. Ce niveau est considéré comme une optimisation dans le cas où le nombre de composants d'un type donnée est égale à 1. On distingue deux sous-niveaux d'optimisation :
P2.1	Le paramètre `this` est ajouté aux méthodes d'une ou plusieurs interfaces serveur du composant, et non ajouté aux méthodes des autres interfaces et aux méthodes privées. Ce niveau est nécessaire si l'on réalise que les assemblages de composants sont susceptibles d'être modifiées, notamment la direction des liaisons. En effet, supposons qu'il existe un composant dans l'application qui expose une interface du même type que celle en question. Dans le cas où ce composant a son code partagé avec d'autres composants du même type (i.e. le niveau d'optimisation concernant le partage de code est de 0 ou 1), une liaison dirigée vers cette interface ne pourrait pas être dirigée vers l'interface du même type dont le code n'est pas partagé. Ce niveau d'optimisation assure donc l'interopérabilité entre interfaces serveur du même type, implémentées par des composants de type diffèrent.
P2.2	Le paramètre `this` n'est pas ajouté à aucune méthode de l'implémentation du composant, serveur ou privée. L'interopérabilité à laquelle on faisait allusion précédemment n'est donc pas assurée. Il est nécessaire donc que les interfaces serveur du composant soient les seules de leurs types dans l'application.

Table 6.13 – Description niveaux d'optimisation du partage de code

Efficacité des optimisations. Nous constatons que les niveaux d'optimisation ici présentés peuvent être classifiés en : i) ceux qui agissent majoritairement sur les méta-données du modèle générées par la chaîne de compilation, et ii) ceux qui

se focalisent sur les interfaces et les composants qui sont ajoutés par la chaîne de compilation et qui sont liés principalement au contrôle de la structure du logiciel. Le tableau 6.14 détaille la correspondance entre les niveaux présentés et ces deux catégories [5].

Catégorie	Niveaux d'optimisation
i) Optimisations sur les méta-données	A4, IS2, L4
ii) Optimisations sur les interfaces et composants de contrôle	A1-A3, IS1, L1-L3, CO1.1, P1

Table 6.14 – Classification des niveaux d'optimisation

En ce qui concerne leur efficacité, nous remarquons que les gains plus importants sont obtenus par celles appartenant au deuxième groupe. Comme l'on verra au chapitre suivant de cette thèse, ces optimisations affectent négativement les capacités d'évolution du système.

Techniques d'optimisation classiques. Nous identifions des nombreuses similarités entre les niveaux d'optimisation proposés et les techniques d'optimisation des plates-formes de référence présentées au chapitre 3.6. Tout d'abord, leurs implémentations mènent majoritairement à la manipulation du code « glue » généré entre les composants et entre le contenu de ceux-ci et les réifications des entités du modèle.

Ensuite, nous retrouvons les techniques d'optimisation des plates-formes de référence présentées au chapitre 3.7 parmi celles définies dans ce chapitre. Le tableau 6.15 résume la correspondance entre les premières et les niveaux d'optimisation proposés. Par ces faits, on infère que les compromis entre critères d'optimisation, et notamment celui entre l'optimisation et l'évolution évoqué précédemment, seront retrouvés dans l'application des niveaux proposés.

Type d'optimisation (cf. 3.7)	Niveaux d'optimisation
Optimisation des connexions	L4, L5, CO1.2
Enlèvement de code non accessible	IS3
Propriétés constantes	A4
Composants multi/mono instanciables	P1, P2

Table 6.15 – Correspondance entre les optimisations étudiées au chapitre 3 et les niveaux d'optimisation proposés.

Considérations sur les activités ultérieures du cycle de vie. Tout comme à partir d'une description de haut niveau d'un composant, on obtient plusieurs réifications à la compilation, il est aussi plausible d'obtenir plusieurs versions du binaire

5. Les niveaux qui provoquent la disparition des entités (e.g. IS3, CO1.2) ne sont pas inclues dans ce tableau.

exécutable en fonction des décisions prises lors de la compilation et du déploiement des composants. Par exemple, nous avons décrit dans ce chapitre comment les effets de l'application des niveaux d'optimisation pouvaient être également obtenus par des opérations menées à la compilation en binaire et au linkage (e.g. dans le cas des interfaces serveur) et comment le résultat et l'efficacité de certaines techniques présentées dépend de ces opérations (e.g. dans le cas des liaisons *inline*). Cela nous indique qu'afin de contrôler la performance des composants à l'exécution, une maîtrise des activités de compilation à bas-niveau et de déploiement est nécessaire.

Par ailleurs, les résultats des actions menées ultérieurement dans le cycle de vie des composants dépendent de la manière dont ces composants à la conception ont été réifiés à la compilation. Dans ce chapitre nous sommes partis des constructeurs par défaut de la chaîne de compilation Fractal/Think, qui cherchent à minimiser le dégrée de dépendance vis-à-vis les compilateurs de bas-niveau. Néanmoins, on pourrait hypothétiquement associer un comportement diffèrent à ces constructeurs pour ainsi ouvrir la porte à des niveaux différents d'optimisation et à nouvelles dépendances entre le compilateur Fractal/Think et les compilateurs de bas-niveau. Dans le cadre de cette thèse nous nous restreindrons aux constructeurs par défaut.

Chapitre 7

Infrastructures d'Évolution

Sommaire

7.1 Évolution dans Fractal/Think 117
7.2 Caractérisation de l'évolution des entités du modèle 120
7.2.1 Évolution dynamique 120
7.2.2 Pas d'évolution 121
7.2.3 Évolution statique 122
7.2.4 Évolution non prédéfinie 125
7.3 Mise en œuvre de l'Évolution non prédéfinie 125
7.3.1 Composants miroir 125
7.3.2 Lien interne entre miroirs et réifications embarquées 133
7.3.3 Réifications embarquées 134
7.4 Infrastructures d'évolution au niveau système 136
7.5 Synthèse 137

Dans ce chapitre, nous présentons l'infrastructure optimisée d'évolution que nous proposons. Nous commençons par identifier les éléments qui peuvent évoluer dans les systèmes modélisés à l'aide de Fractal/Think. Ensuite, nous définissons quatre catégories d'évolution applicables à ces entités.

Puis, nous proposons des infrastructures d'évolution adaptées et optimisées pour les entités selon leur catégorie d'évolution. Leur implémentation repose sur les optimisations décrites au chapitre 6. Pour l'une d'elles, nous concrétisons les concepts définis au chapitre 5, raison pour laquelle nous lui consacrons un chapitre. Enfin, nous énonçons les considérations à prendre en compte lorsqu'on considère des catégories multiples d'évolution pour les entités d'un système.

7.1 Évolution dans Fractal/Think

Les infrastructures d'évolution (IdE) prévues par défaut par la plate-forme Fractal/Think définissent des points précis d'observation de l'état des composants et d'intervention sur la structure et le comportement lié à ceux-ci. Les actions d'évolution sont implémentées par des interfaces offertes par les composants, parmi lesquelles on retrouve les méthodes des interfaces de contrôle présentées au chapitre précèdent, et dont les fonctionnalités offertes sont décrites dans le tableau 7.1.

Les implémentations de ces interfaces dépendent de la génération des métadonnées sur les entités du modèle à composants qui sont, soit exclusives à l'aspect évolution, soit partagées avec les aspects fonctionnels ou d'autres aspects extra-fonctionnels des composants.

Interface	Méthodes Déclaration	Description
	Interfaces orientées réflexion - introspection	
`ComponentContent`	`Object getSubComponent(String name)`	Récupérer un descripteur d'un sous-composant par son nom
	`int getNumOfSubComponents()`	Récupérer le nombre de sous-composants
	`List<String> listSubComponents()`	Récupérer les noms des sous-composants
`ComponentIdentity`	`Object getInterface(String name)`	Récupérer le descripteur d'une interface par son nom
	`List<String> getInterfaces()`	Récupérer les noms des interfaces
	`String getInterfaceType(String name)`	Récupérer le type de l'interface par son nom
	`int getNumOfInterfaces()`	Récupérer le nombre d'interfaces
	Interfaces orientées réflexion - intercession	
`AttributeController`	`void setAttValue(String name, Object value)`	Donner une valeur à l'attribut `name`
	`Object getAttValue(String name)`	Récupérer la valeur de l'attribut `name`
	`List<String> listAttributes()`	Récupérer la liste des noms des attributs
	`int getNumOfAttributes()`	Récupérer le nombre d'attributs
`BindingController`	`void bind(String cltItfName, Object srvItf)`	Créer une liaison entre l'interface cliente `cltItfName` et l'interface serveur identifiée par le descripteur `srvItf`
	`void unbind(String clientItfName)`	Détruire la liaison où l'interface `clientItfName` est cliente
	`List<String> list()`	Récupérer les noms des interfaces clientes
	`Object lookup(String clientItfName)`	Récupérer le descripteur de l'interface serveur liée à l'interface cliente `clientItfName`, ou la valeur `null` si elle n'est pas liée à aucune interface
`ContentController`	`addSubComponent(String compName, Object subCompId)`	Ajouter un sous-composant `subCompId` sous le nom `compName`
	`Object removeSubComponent(String compName)`	Enlever le sous-composant dont le nom est `compName`
	Interfaces de gestion du cycle de vie	
`LifeCycleController`	`void start()`	Démarrer l'execution du composant
	`void stop()`	Arrêter l'execution du composant
`Reconf.Controller`	`int suspend(Object compId)`	Suspendre l'execution du composant
	`int resume(Object compId)`	Reprendre l'execution du composant
	`int getState()`	Récupérer l'état du composant

TABLE 7.1 – Interfaces offertes par les composants Fractal/Think. Ces interfaces font partie de leur IdE par défaut.

Ainsi, l'évolution dans Fractal/Think s'appuie sur cinq groupes d'interfaces :

- **Interfaces orientées réflexion - introspection**. Les interfaces d'intros-

pection (interfaces `ComponentIdentity` et `ComponentContent` [1]) permettent l'observation des caractéristiques des composants, telles que les signatures des méthodes offertes ou les sous-composants.

- **Interfaces orientées réflexion - intercession**. Les interfaces d'intercession (interfaces `AttributeController`, `BindingController` et `ContentController`) permettent la modification de la structure interne et du comportement des composants, via la redirection ou la création des liaisons entre composants, l'ajout ou l'enlèvement de sous-composants et le changement de la valeur d'un attribut.
- **Méta-données propres aux interfaces de réflexion**. Les méta-données associées aux interfaces de réflexion, i.e., celles qui sont générées par la chaîne de compilation pour permettre l'exécution des méthodes des interfaces de réflexion. Au chapitre précèdent, on a montré la façon dont ces méta-données participent au coût associé à ces interfaces.
- **Méta-données sur les entités du modèle**. Ces méta-données sont générées par la chaîne de compilation. Les interfaces liées à l'évolution (ou les méta-données associées), mais aussi les interfaces purement fonctionnelles ou celles liées à des aspects autres que l'évolution, font usage de ces méta-données. Par exemple, au chapitre 6.2, la structure `attsDesc` est référencée à chaque accès à un attribut lorsque celui-ci est compilé avec un niveaux d'optimisation entre A0 et A3, indépendamment d'où, dans le contenu du composant, cet accès est effectué .
- **Interfaces de gestion du cycle de vie**. Ces interfaces (`LifeCycleController` et `ReconfigurationController`) offrent un contrôle sur l'état d'opération du composant et permettent de (ré) démarrer, suspendre, reprendre et arrêter son exécution. Contrairement aux autres éléments de l'infrastructure d'évolution, les interfaces de gestion du cycle de vie font référence au composant en sa totalité et non de manière individuelle aux entités du modèle.

Les premiers quatre éléments sont nécessaires afin d'effectuer les actions d'évolution. Le cinquième est utilisé pour assurer la fiabilité de ces actions, dans la mesure où certaines d'entre elles peuvent demander la suspension de l'exécution d'un composant ou même la transition d'un état du composant à un autre. L'implémentation des méthodes de gestion du cycle de vie dépend de la politique choisie pour assurer la conservation d'un état stable du composant. Dans le cadre de cette thèse nous ne traitons pas cet aspect de la fiabilité des actions d'évolution, nous orientons le lecteur vers [Polakovic 2008] pour une étude exhaustive sur ce sujet.

Dans Fractal/Think, un composant externe à l'application ou l'application elle même, peut appeller les interfaces liées à l'évolution. Il n'y a pas de différence de mise en œuvre selon qui est l'appelant et quand l'appel est fait.

1. L'interface `ComponentContent` contient des méthodes qui ont été présentées au chapitre 6 comme faisant partie de l'interface `ContentController`. Nous distinguons ici les deux interfaces afin d'afficher clairement la nature distincte des deux groupes de méthodes : alors que la première concerne l'introspection de composants, la deuxième concerne la modification de l'architecture logicielle.

Les considérations d'évolution et la mise en œuvre restent générales. Dû au compromis entre richesse d'évolution et performance du logiciel, on constate des surcoûts importants, notamment sur l'occupation mémoire du logiciel, que nous jugeons non justifiés lorsque non nécessaires [Navas 2009b].

Afin d'améliorer les performances, nous présentons maintenant les catégories d'évolution que nous proposons d'associer aux entités du modèle à composants, catégories utilisées pour diriger la mise en œuvre des infrastructures d'évolution.

7.2 Caractérisation de l'évolution des entités du modèle

Dans l'état de l'art, nous avons identifié le compromis existant entre la richesse d'évolution fournie par les IdE, et l'impact des IdE sur la performance du logiciel. Ce compromis a été par ailleurs constaté sur l'optimisation, dans la mesure où les optimisations plus efficaces sont justement celles qui concernent les interfaces de réflexivité, qui font partie des IdE.

En conséquence, nous proposons quatre catégories d'évolution (plus ou moins forte) qui peuvent être appliquées aux entités du modèle à composants. Ces catégories seront utiles pour guider la génération de code optimisée. Elles sont définies par rapport aux caractéristiques d'évolution souhaitées pour les entités du modèle, concrètement en réponse aux questions suivantes :

- Est-ce que l'entité évoluera après le déploiement du logiciel ? Cette information est-elle disponible et si oui, à quelle étape du cycle de vie ?
- Si l'entité va évoluer, comment l'entité évolue ? Quelles sont ses politiques d'évolution ? Quelles sont les valeurs qu'elle peut prendre ?
- Qui déclenche les actions d'évolution qui concernent l'entité ?
- A quel moment de l'exécution du logiciel l'entité évoluera ?

En fonction des réponses à ces questions, nous proposons quatre catégories d'évolution que nous détaillons maintenant. Les chapitres suivants sont consacrés à la définition et à la description de la mise en œuvre des IdE qui correspondent aux quatre catégories proposées. Nous proposons notamment des IdE optimisées « sur mesure » pour chacune d'elles. Le tableau 7.2 synthétise les réponses de ces quatre catégories aux question posées ici.

7.2.1 Évolution dynamique

Description. A ce niveau, le fait que l'entité va évoluer à l'exécution est connu dès les premières étapes du développement du système, i.e. à la conception et au développement. Toutefois, les détails sur la manière dont elle évoluera, i.e. les politiques d'évolution, ne sont pas connus à ce moment-là ou sont très difficiles à estimer à priori (comportement complexe des évolutions). Aucune restriction n'est posée par rapport au déclenchement des actions d'évolution, ni au moment où l'entité évoluera. Par conséquent, le système doit fournir lui-même la possibilité d'exécuter toutes les actions d'évolution possibles qui concernent l'entité, à n'importe quel moment et par n'importe quel acteur.

	Dynamique	Pas d'évolution	Statique	Non prédéfinie
L'entité évoluera après déploiement ?	Oui	Non	Oui	Oui
Quand on le sait ?	Conception	Conception	Conception	Conception/ Exécution
On sait à la conception comment elle évoluera ?	Non	-	Oui	Non
Qui déclenche les actions d'évolution ?	Les composants du système et/ou des composants externes	-	Les composants de l'application	Des composants externes
Quand ces actions sont déclenchées ?	On sait pas prédire	-	On sait prédire	On sait pas prédire

TABLE 7.2 – Questions posées lors de la définition des catégories d'évolution à assigner aux entités du modèle.

Mise en œuvre. Ce niveau correspond à celui par défaut dans l'état actuel de la plate-forme Fractal/Think, où l'évolution est dirigée via les interfaces de réflexion (introspection et intercession).

Par rapport aux niveaux définis au chapitre précédent, l'entité est réifiée au niveau d'optimisation 0 comme l'illustre le tableau 7.3. L'interface de réflexivité correspondante est ajoutée au contenu du composant : `AttributeController` pour les attributs, `BindingController` pour les liaisons, `ComponentIdentity` pour les interfaces serveur et `ContentController` et `ComponentContent` pour les composites. Les méta-données associées à ces interfaces de réflexivité sont également générées.

Ce niveau est le niveau par défaut dans Fractal/Think, où toutes les entités du modèle peuvent évoluer. Il peut être vu comme la référence : la génération de code produit un code certes lourd, mais riche en alternatives d'évolution.

Entité	Optimisation
Attributs	A0
Interfaces serveur	IS0
Liaisons	L0
Composites	CO0
Composants (mono/multi instance)	P0

TABLE 7.3 – Niveau d'optimisation assigné aux entités dans la catégorie Évolution Dynamique.

7.2.2 Pas d'évolution

Description. Ce niveau correspond au cas où l'entité du modèle à composant n'est pas plausible d'évoluer après le déploiement du logiciel, et où cette information est connue à la phase de conception de celui-ci.

Mise en œuvre. L'infrastructure d'évolution qui concerne l'entité n'est pas générée par la chaîne de compilation. Au cas où l'évolution et l'optimisation sont les seuls aspects extra-fonctionnels pris en compte, ou que les autres aspects considérés permettent le faire, les méta-données sur l'entité ne sont pas non plus générées. La mise en œuvre de ce cas correspond donc au niveau maximum d'optimisation appliquée à l'entité, comme il est décrit par la suite et illustré dans le tableau 7.4.

- *Attributs* : ce niveau est limité aux attributs *constants*, i.e. dont la valeur figée à la conception ne change pas à l'exécution. Le niveau d'optimisation A4, où les références à la valeur de l'attribut sont remplacées par une valeur constante équivalente à celle définie à la conception, typiquement via une macro (e.g. `#define`), est donc appliqué aux attributs.
- *Liaisons* : le niveau L4, où les liaisons sont réifiées en simples appels statiques à procédures, est appliqué par défaut. Compte tenu du compromis entre les critères d'optimisation occupation mémoire et temps d'exécution, le niveau L5 est appliqué seulement si cela est justifié et si explicitement indiqué à la conception du système. Comme présenté au chapitre 6.5, la décision finale d'*inliner* une méthode est prise par le compilateur de bas niveau.
- *Interfaces serveur* : le niveau IS2, ou les méta-données sur l'interface serveur ne sont pas générées, est appliqué par défaut. Au cas où à la conception du système on remarque qu'aucune liaison est dirigée vers l'interface, le niveau IS3, où les définitions des méthodes de l'interface ne sont pas ajoutées au contenu du composant, est appliqué. Un résultat similaire est obtenu si on applique aux liaisons dirigées vers l'interface le niveau d'optimisation L5 et si le compilateur de bas niveau effectue l'*inline* des méthodes. Mais cela reste dépendant des décisions prises par le compilateur.
- *Composites* : les composites qui n'évoluent pas après déploiement sont optimisés au niveau CO1.2, ce qui signifie la non génération des interfaces et des liaisons de délégation et par conséquent la disparition du composite, i.e. la non génération de réification embarqué pour celui-ci.
- *Composants mono/multi instance* : compte tenu du compromis évoqué au chapitre 6.7, les trois niveaux d'optimisation concernant le partage de code entre composants du même type sont appliqués en fonction de l'architecture logicielle. Si le nombre de composants d'un type donné est supérieur à 1, P1 est appliqué. Dans le cas contraire P2.1 est appliqué. Le niveau P2.2 est appliqué si en plus de cela le type des interfaces serveur du composant est instancié une seule fois dans l'application.

A l'inverse du cas précèdent, ce niveau est le niveau d'optimisation maximal, sans évolution possible. C'est le cas typique lorsque l'intérêt n'est porté que sur l'optimisation de l'utilisation des ressources physiques de la plate-forme.

7.2.3 Évolution statique

Description. A ce niveau on sait que l'entité évoluera à l'exécution, et on connaît à la conception les politiques de son évolution. Par conséquent, l'ensemble

Entité	Optimisation
Attributs	A4
Interfaces serveur	IS2, IS3 si pas de liaisons vers l'interface
Liaisons	L4, L5 si justifié ou indiqué explicitement
Composites	CO1.2
Composants (mono/multi instance)	P1 ou P2.1 selon le cas, P2.2 si possible

TABLE 7.4 – Niveau d'optimisation assigné aux entités dans la catégorie "Pas d'évolution".

d'actions d'évolution que la IdE sera amenée à effectuer après son déploiement est connu et défini à priori, de sorte que des techniques d'optimisation à la compilation peuvent être appliquées en fonction de celles-ci. Ces actions sont déclenchées par le composant associé à l'entité ou pour un intergiciel liée à celui-ci. Le moment précis de l'exécution du système où les actions d'évolution sont déclenchées est en principe connu lui aussi, puisqu'il ressort de l'analyse du comportement des composants. Ce cas correspond à la définition, au moment de la conception, de modes d'opération du système (dont on a fait référence au chapitre 4.4.1.1).

Mise en œuvre. Dans cette catégorie, les niveaux d'optimisation appliqués sur les entités varient principalement en fonction de (i) qui déclenche les actions d'évolution associées à l'entité, et plus concrètement s'il s'agit du composant lié à l'entité, ou d'un autre composant ou intergiciel externe à celui-ci, mais toujours embarqué dans la même plate-forme matérielle, et de (ii) la connaissance approfondie sur le comment et le quand des actions d'évolution. Le tableau 7.5 résume les niveaux d'optimisation que l'on peut appliquer aux entités ; nous les détaillons par la suite.

- *Attributs* : si c'est le composant lié à l'attribut qui déclenche les actions d'évolution, le niveau A3, où les méta-données sur l'attribut sont générées, est appliqué. Si c'est un autre composant où un intergiciel, le niveau A0 est appliqué puisque l'interface `AttributeController` doit être ajoutée au contenu du composant. Dans ce dernier cas on peut considérer des optimisations telles que l'*inline* des méthodes de cette interface dans le contenu du composant appelant (cf. liaisons).
- *Interfaces serveur* : le modèle Fractal/Think n'offre pas l'ajout ou la suppression d'interfaces serveur mais seulement leur introspection. L'interface `ComponentIdentity` doit être ajoutée au contenu du composant associé à l'interface, ce qui signifie que le niveau d'optimisation IS0 est appliqué. Par contre on peut considérer des optimisations telles que l'*inline* des méthodes de cette interface dans le contenu du composant appelant (cf. liaisons).
- *Liaisons* : en principe l'interface `BindingController` doit être ajoutée au contenu du composant, ce qui implique un niveau d'optimisation L0. Néanmoins, une connaissance approfondie sur la manière dont la liaison évoluera dans le temps, permet de se passer de l'inclusion des méthodes d'introspection (interface `ComponentIdentity`) dans le contenu des composants exposant l'interface serveur vers laquelle la liaison sera dirigée dans le futur, permet-

tant des optimisations autres que IS0 concernant ces interfaces. Également, en fonction des caractéristiques structurales et comportementales des composants on peut appliquer le niveau d'optimisation L5 (*inline*) sur des liaisons où cela soit pertinent, y compris celles dirigées vers des interfaces qui constituent les infrastructures d'évolution.

- *Composites* : les interfaces `ComponentContent` et `ContentController` qui fournissent des services d'introspection et intercession concernant les sous-composants d'un composite, respectivement, doivent être ajoutées au contenu du composite. Cela implique un niveau d'optimisation CO0. Par contre, la connaissance détaillée de leur évolution (e.g. le nombre maximal de sous-composants) détermine des optimisations possibles concernant l'implémentation de ces méthodes. Également, on peut considérer des optimisations telles que l'*inline* des méthodes de ces interfaces dans le contenu du composant appelant (cf. liaisons).

- *Composants (mono/multi instance)* : l'établissement d'une borne maximale au nombre d'instances d'un type de composant ou la connaissance précise du nombre de ces instances, détermine des optimisations possibles concernant l'implémentation des composants de type `Factory` liés au type de composant, ou la non inclusion de ce composant au système. Dans ces cas, et concernant les optimisations proposées au chapitre précèdent, les niveaux P1 et P2.1/2.2 sont appliqués, respectivement.

Ce cas pourrait être traité comme le premier cas ("Évolution dynamique"). Mais, on constate que des optimisations peuvent être appliquées sur les entités de cette catégorie en se basant sur une analyse exhaustive du comportement des composants et de celui lié aux IdE, et moins sur les aspects purement structurels des assemblages des composants. La plate-forme Fractal/Think ne dispose pas des outils nécessaires pour mener une telle analyse sur les implémentations des composants en langage C, par exemple à base de techniques d'analyse statique de code. Par conséquent, cette catégorie n'est pas traitée dans cette thèse. Ce cas correspond aux domaines où l'optimisation du code dépend très fortement de l'étude de son comportement. Nous en reparlerons au chapitre 9 de ce manuscrit.

Entité	**Optimisation**
Attributs	A0, A3
Interfaces serveur	IS0
Liaisons	L0, L5
Composites	CO0
Composants (mono/multi instance)	P1, P2.1/2.2

TABLE 7.5 – Niveau d'optimisation assigné aux entités dans la catégorie Évolution Statique.

7.2.4 Évolution non prédéfinie

Description. A ce niveau, le fait que l'entité *peut* évoluer est connu à la conception. Par contre, le fait qu'elle *va* évoluer est connu après le déploiement du composant. Les politiques d'évolution concernant l'entité ne sont donc pas connues lors de la conception et du développement du logiciel. L'évolution n'est pas dirigée par l'application mais elle est dirigée par un agent externe. Ce cas correspond à des activités typiques de l'évolution et de la maintenance logicielle telles que la syntonisation (*tuning*) de paramètres, la correction de *bugs* et le raffinement du comportement du système.

Mise en œuvre. La mise en œuvre de ce cas d'évolution dans Fractal/Think est la même que pour le cas d'évolution dynamique (cf. 7.2.1), même si les deux catégories n'ont pas les mêmes propriétés, ni ne répondent aux mêmes exigences.

La mise en œuvre des trois catégories précédentes repose entièrement sur l'application des niveaux d'optimisation proposées au chapitre 6 de cette thèse, la mise en œuvre de celle-ci utilise et implante les notions de réification et notamment de composant miroir, développées au chapitre 5. Nous consacrons donc le chapitre suivant à l'étude détaillée de la mise en œuvre de cette catégorie.

7.3 Mise en œuvre de l'Évolution non prédéfinie

L'infrastructure d'évolution proposée pour cette catégorie d'évolution se base sur l'idée, d'une part, de débarquer une partie de la IdE pour ainsi traiter un sous-ensemble d'aspects liées à l'évolution du logiciel dans un environnement plus riche et moins contraint au niveau des ressources physiques que le dispositif embarqué, et d'une autre part, d'optimiser l'IdE embarquée.

A cette fin, nous implantons les concepts définies au niveau des réifications, notamment ceux liés aux composants miroirs (cf. chapitre 7.3.1). Pour assurer la correspondance entre les composants miroirs et la réification du composant embarquée dans le dispositif, un ensemble des réifications embarquées et débarquées, détaillées au chapitre 7.3.2, sont crées à l'étape de compilation du système. Cette démarche nous permet d'appliquer des niveaux d'optimisation sur les réifications embarquées sans affecter la richesse d'évolution, comme il est décrit au chapitre 7.3.3.

Nous commençons par présenter les réifications *off-site*, puis les réifications embarquées ou *in-site*. Nous restons ainsi dans le sens des actions d'évolution suggéré par cette catégorie d'évolution : déclenchées par des agents externes au dispositif, dans ce cas dans la plate-forme *off-site* via le composants miroir.

7.3.1 Composants miroir

Les composants miroir sont à la base de l'IdE, dans la mesure où ils encapsulent des aspects liées à l'évolution logicielle. Pour chaque composant inclus dans la description du système à la conception, la chaîne de compilation génère un composant miroir comme réification à l'exécution, en plus de la réification embarquée considérée tout au long de ce chapitre. Nous parlerons ici des composants miroir en utilisant

le vocabulaire propre des composants et plus spécifiquement du modèle Fractal. Ils sont implémentés comme des objets Java.

Les composants miroir offrent un nombre de services regroupés dans ses interfaces serveur, qui sont invoqués soit par d'autres composants miroir, soit par des consoles d'administration et de maintenance. Afin de rendre les services d'évolution, les composants miroir doivent avoir une vue sur le cycle de vie des composants et pouvoir modifier les réifications embarquées des composants, i.e. l'implémentation mémoire. Ces exigences sont modélisées par des liens horizontaux à partir de ses interfaces clientes (cf. 7.3.1.2) et par des liens verticaux avec les réifications du composant à d'autres étapes de son cycle de vie, afin de récupérer des informations à propos de la manière dont les réifications embarquées ont été compilées et déployées dans le dispositif (cf. 7.3.1.3).

7.3.1.1 Services offerts par les composants miroir

Les interfaces de type serveur offertes par les composants miroir sont celles qui font partie de l'infrastructure d'évolution classique Fractal/Think (cf. 7.1) : les interfaces orientées réflexion (introspection et intercession) et celles liées à la gestion du cycle de vie, dont les descriptions ont été présentées dans le tableau 7.1.

En ce qui concerne l'implémentation de leurs méthodes :

- Pour les méthodes des interfaces de gestion du cycle de vie (`LifeCycleController` et `ReconfigurationController`), le comportement consiste à faire appel aux méthodes de ces interfaces qui sont fournies par les réifications embarquées, si elles ont été effectivement rajoutées au contenu des composants ; sinon, le comportement par défaut consiste à renvoyer une erreur indiquant l'absence de réification embarquée de ces interfaces.
- Pour les méthodes des interfaces d'intercession (BindingController, `ContentController` et `AttributeController`), le comportement consiste à exécuter les actions d'intercession correspondantes dans le dispositif embarqué. Pour cela les composants miroir requièrent un ensemble de services qui seront fournis par d'autres composants et qui seront détaillés au chapitre 7.3.1.2.
- Pour les méthodes des interfaces d'introspection (`ComponentIdentity`, `ComponentContent`) et la méthode `lookup` de l'interface `BindingController`, le comportement consiste à fournir l'information demandée à partir de l'analyse de la façon dont les réifications embarquées ont été construites lors des étapes de compilation, déploiement et activation. Cela se fait en forgeant des liens verticaux avec d'autres réifications du composant, ce qui est détaillé au chapitre 7.3.1.3.

Les actions d'optimisation menées à la compilation des composants déterminent en grande mesure le comportement lié à ces interfaces. En effet, ces optimisations peuvent mener à la disparition de certaines entités du modèle à l'exécution. Par exemple, pour un attribut auquel le niveau A4 a été appliqué, sa valeur est toujours accessible (i.e. la méthode `getAttribute` de l'interface `AttributeController` de son

miroir fournira la valeur définie à la conception) mais l'entité a été remplacée par une valeur constante non modifiable. Dans ces cas, l'appel aux méthodes correspondants (dans l'exemple, la méthode `setAttribute` de l'interface `AttributeController`) renvoient une erreur.

7.3.1.2 Services requis par les composants miroir

Les interfaces requises des composants miroirs sont listées dans le tableau 7.6.

Interface	Méthodes Déclaration	Description
`MemoryController`	`byte[] read(BinAddress add, int size)`	Lire des plages de mémoire physique déterminées par l'adresse `add` et la taille `size` en octets
	`void write(BinAddress add, int size, byte[] data)`	Ecrire des données `data` dans la plage de mémoire physique définie par l'adresse initiale `add` et taille `size` en octets
`EmbeddedLCC`	`void setComponent(BinSymbol srvItf)`	Définir lequel des composants embarqués sera concerné par les méthodes `start` et `stop` de l'interface
	`void start()`	Démarre l'exécution du composant embarqué défini via la méthode `setComponent`
	`void stop()`	Arrête l'exécution du composant embarqué défini via la méthode `setComponent`
`EmbeddedRC`	`void setComponent(BinSymbol srvItf)`	Définir lequel des composants embarqués sera concerné par les méthodes `suspend` et `resume` de l'interface
	`int suspend()`	Suspendre l'exécution du composant
	`int resume()`	Reprendre l'exécution du composant
	`int getState()`	Récupérer l'état du composant

TABLE 7.6 – Interfaces requises par les composants miroir.

`MemoryController`. Les méthodes de cette interface permettent la lecture et l'écriture des zones de mémoire physique dans le dispositif embarqué. Elles reçoivent comme premier et deuxième paramètre une adresse de base et la taille de la zone mémoire en octets. Les composants miroirs utilisent ces interfaces dans l'implémentation des méthodes d'intercession.

L'adresse base est un objet de la classe BinAddress dont une description est fournie par la figure 7.1. Elle permet de connaître l'emplacement mémoire de l'entité manipulée. Une adresse mémoire est caractérisée ainsi par les attributs suivants :

- Une variable `segment` qui sauvegarde le nom du segment mémoire auquel l'adresse appartient. Ce nom correspond au type de mémoire, typiquement ROM, FLASH ou RAM. Le nom `unknown` est assigné si le segment n'est pas connu.
- Une variable `shadow` qui a une valeur true si la valeur de l'adresse a été obtenue à partir d'un processus de shadowing (cf. 2.1.2.1) d'une autre adresse correspondant à un autre segment, et false dans le cas contraire.
- Une variable `address` qui contient la valeur numérique d'une adresse mémoire (e.g. `0x00004056`).

– Une variable `offset` qui permet de trouver la valeur numérique effective de l'adresse mémoire représentée par BinAddress, spécifiée par `addresse+offset`. La valeur d'offset est donnée en un numéro entier de Bytes. Cette variable est utile lorsque l'adresse en question est un champ d'une structure de données plus complexe.

Nous remarquons qu'une BinAddress contient des informations sur l'effet de la compilation, plus précisément sur l'effet du déploiement du système dans la mémoire du dispositif. La façon dont ces objets BinAddress sont obtenus par le composant miroir est détaillée au chapitre 7.3.1.3.

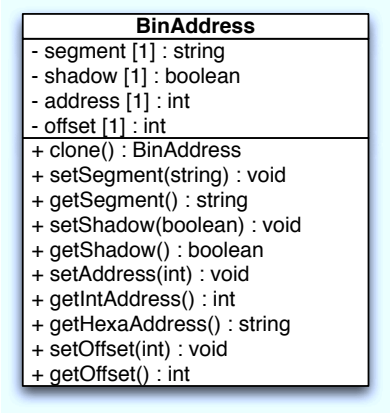


FIGURE 7.1 – Classe BinAddress.

`EmbeddedLCC`. Le type de cette interface correspond à peu près à celui de l'interface `LifeCycleController` dans le tableau 7.1, qui est offerte par le composant miroir. Typiquement les implémentations des méthodes de l'interface `LifeCycleController` feront appel aux méthodes correspondantes de l'interface cliente `EmbeddedLCC`. A son tour, l'invocation à ces méthodes permet d'effectuer des invocations aux méthodes correspondantes de l'interface qui est implémentée par les réifications embarquées du composant. Cela permet d'assurer que l'état de la réification embarquée correspond à l'état de la réification miroir [2].

Une méthode `setComponent` est ajoutée à cette interface. Cette méthode **doit être invoquée avant toute invocation** aux autres méthodes de l'interface `EmbeddedLCC`. Elle permet de fixer la réification embarquée sur laquelle les opérations de gestion du cycle de vie seront propagées. Le paramètre d'entrée `lccSrvItf` est un objet de la classe BinSymbol qui représente la réification de l'interface `LifeCycleController` au sein de la réification embarquée.

BinSymbol est utilisé pour construire une correspondance entre les entités du modèle à la conception et les symboles binaires à l'exécution, ce qui sera illustré au

2. Compte tenu de la propriété 5.2.2.v.a, les états des deux réifications doivent être consistants. Cela est assuré dans l'implémentation de la méthode `getState` de l'interface serveur `LifeCycleController` du composant miroir, qui obtient l'état courant de la réification embarqué vie l'interface cliente `MemoryController`.

chapitre 7.3.1.3. C'est donc un lien entre plusieurs appellations (à des niveaux de réification différents) et les emplacements mémoire correspondants.

La figure 7.2 présente la classe BinSymbol qui possède les attributs suivants :

- Une variable `name` qui identifie le symbole dans le contexte du composant miroir. Par exemple, pour l'interface serveur `sound` du composant `sound_driver` dans la figure 6.4 ce nom serait `itf_srv_sound`.
- Une variable `binSymName` qui correspond au nom du symbole binaire défini à la conception ou assigné par la chaîne de compilation. Par exemple, dans la figure 6.4 le nom du symbole créé par la chaîne de compilation pour la méthode `beep_async` de l'interface `sound` du composant `sound_driver` est `simple_sound_driver__default__SRV_sound__beep_async`.
- Une variable `addresses` qui est une liste de BinAddress contenant les adresses mémoire qui sont liées au symbole. Typiquement elle contiendra une adresse en ROM ou FLASH et une adresse shadow en RAM.

Tout comme pour les objets BinAddress, la façon dont ces objets `BinSymbols` sont obtenus par le composant miroir est détaillée au chapitre suivant.

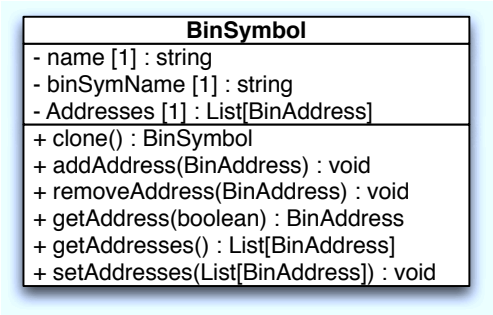


FIGURE 7.2 – Classe BinSymbol.

7.3.1.3 Récupération des informations sur la compilation et le déploiement des réifications embarquées

Comme cela a été évoqué aux deux chapitres précédents, afin d'effectuer des opérations liées à l'évolution des réifications embarquées, les composants miroirs doivent avoir une connaissance approfondie sur la manière dont leurs homologues embarqués ont été compilés et déployés dans la plate-forme matérielle. Cette information est récupérée au travers des actions menées pendant tout le cycle de vie du composant, particulièrement lors de sa compilation et de son déploiement.

Les réifications établissent des liens du type vertical (cf. 5.2.3.1) où la réification associée à l'étape plus avancée du cycle de vie requiert un service à la réification de l'étape précédente, comme l'illustre la figure 7.3. Ainsi, le composant miroir obtient des informations sur l'activation du composant au travers de la réification associée à l'activation, qui elle-même se base sur l'information fournie par la réification au déploiement, et ainsi de suite.

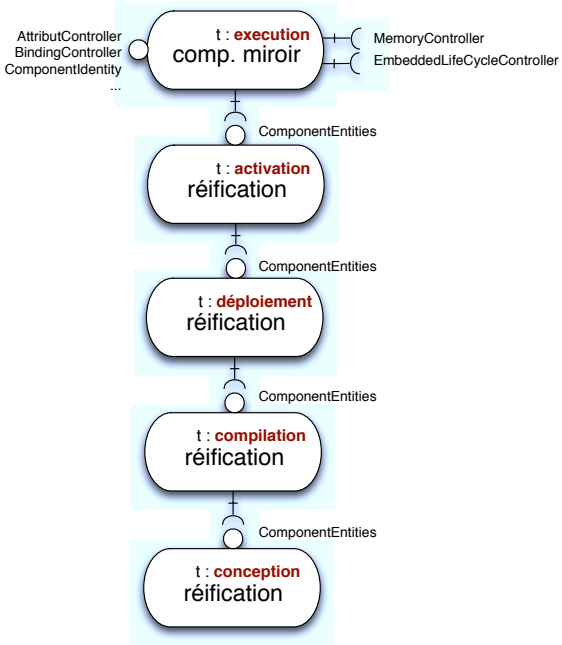


FIGURE 7.3 – Liens verticaux entre réifications d'un composant.

Représentation des entités du modèle à composants. Les liens verticaux évoqués précédemment connectent des interface de type `ComponentEntities`, qui comme illustré dans le tableau 7.7, ne contienne qu'une seule méthode, `getEntities`.

	Méthodes	
Interface	**Déclaration**	**Description**
`ComponentEntities`	`Entity[] getEntities(void)`	Recupérer les entités du composant

TABLE 7.7 – Interface `ComponentEntities`.

Cette méthode renvoie une liste d'objets de la classe Entity, caractérisée par :
- une variable `type` qui définit le type d'entité auquel il fait référence (attribut, interface serveur, interface cliente, liaison, composite ou primitif) ;
- une variable `name` qui correspond au nom de l'entité définie à la conception ;
- une variable `symbols` qui est l'ensemble d'objets de classe BinSymbol liés à la réification embarquée de l'entité.

La figure 7.4 présente cette classe et ses relations avec les classes précédemment définies BinSymbol et BinAddress.

Nous associons à chaque entité du modèle Fractal/Think un objet de la classe Entity. La nature de ce lien, entre des abstractions propres à la conception du système et le code binaire en exécution, dépend fortement de la construction de la réification

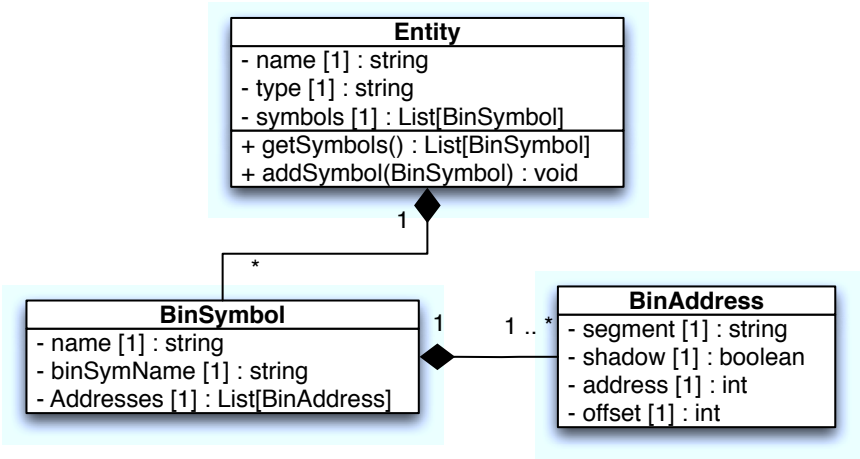


FIGURE 7.4 – Classe Entity.

embarquée des entités, et notamment des optimisations effectuées à la compilation.

Une entité peut « disparaître » au cours du cycle de vie du composant : c'est le cas pour les interfaces serveur non liées lorsqu'on lui applique un niveau d'optimisation IS3, de même, un attribut réifié comme une valeur constante avec un niveau d'optimisation A4, dans la mesure où aucun symbole binaire est générée pour stocker sa valeur. Dans ces cas, l'entité n'a pas de symbole associé.

Pour chaque entité, les adresses binaires auxquelles l'objet Entity fait référence, sont :

- *Attributs* : l'adresse du champ qui correspond à l'attribut dans la structure `attsDesc` (cf. 6.2) du composant.
- *Interfaces serveur* : l'adresse du champ qui correspond à l'interface dans la structure `srvItfsDesc` (cf. 6.3) du composant qui offre l'interface.
- *Interfaces clientes* : l'adresse du champ qui correspond à l'interface dans la structure `cltItfsDesc` (cf. 6.3) du composant qui requiert l'interface.
- *Liaisons* : (i) l'adresse du champ qui correspond à l'interface cliente associée à la liaison dans la structure `cltItfsDesc` du composant client, et (ii) l'adresse du champ qui correspond à l'interface serveur associée à la liaison dans la structure `srvItfsDesc` du composant serveur.
- *Contenu* : l'adresse du symbole `implemDesc` du composant.

Description du comportement associé aux liens verticaux des réifications. Nous décrivons maintenant les opérations menées par les entités associées aux étapes du cycle de vie des composants, en ce qui concerne l'interface `ComponentEntities`.

- *Réification à la conception* : cette réification, qui est étroitement liée aux fichiers qui décrivent le composant à cette étape du cycle de vie (ADL, IDLs

et C), est celle qui crée les objets Entity en fonction des descriptions fournies par les concepteurs, en assignant le type correspondant et le nom défini par ceux-ci.

- *Réification à la compilation* : cette réification récupère les objets Entity du composant de la réification à la conception et crée les objets BinSymbol qui sont associées à l'entité. Ces symboles sont obtenus à partir des opérations menées à la compilation : la génération des méta-données associées à chaque entité, et la transformation du contenu fourni par les concepteurs. Au cas où le symbole associé à l'entité est un champ d'une structure plus complexe, cette réification crée une adresse *dummy* avec la valeur d'offset appropriée. Également, cette réification est chargée de supprimer les entités disparues suite à des opérations d'optimisation.
- *Réification au déploiement* : puisque c'est au déploiement qu'on construit la correspondance entre les symboles du code C et la mémoire physique du dispositif, pour chaque symbole lié à chaque entité associée au composant cette réification lui attribue au moins un objet BinAddress. L'information concernant l'adresse est obtenue en deux étapes : (i) à partir des informations obtenues à la compilation, notamment les noms des symboles et les offsets si y en a lieu, les valeurs numériques des adresses sont obtenues à partir de l'analyse du code binaire du système ; (ii) cette information est corrélée avec un *mapping* de la mémoire physique fourni par les concepteurs, qui a été précédemment utilisé pour générer dynamiquement le script de *linkage*, qui contient des informations concernant les segments et les sections de la mémoire physique. La mise en œuvre de ces opérations est fortement dépendante des compilateurs de bas niveaux et des plates-formes matérielles.
- *Réification à l'activation* : cette réification exécute des actions similaires à la précédente. Néanmoins, les adresses obtenues ici ne font pas référence au déploiement du composant mais aux plages mémoires associées à celui-ci lorsqu'il a été activé et prêt à être exécuté. Typiquement, c'est dans cette étape où les adresses issues du processus de *shadowing* sont obtenues. Les objets Entity fournis au composant miroir, représentent la réification embarquée du composant juste avant son exécution.

Exemple. La figure 7.5 illustre le traitement de l'objet Entity correspondant à l'attribut `maxFreq` du composant `sound_driver`. On remarque comment les informations relatives à l'entité sont remplies progressivement par les réifications du composant `sound_driver` à la conception, à la compilation, au déploiement et à l'activation. L'adresse associée au segment RAM est une adresse de type *shadow*, i.e. sa valeur initiale est une image de l'adresse associée au segment FLASH. Dans ce cas particulier, l'adresse contenant la valeur de l'attribut `maxFreq` n'est associée directement à aucun symbole binaire, mais au symbole `sound_compDesc`. L'adresse du symbole de `maxFreq` a été obtenue en ajoutant un offset de 32 Bytes à l'adresse de ce symbole.

Hypothèse. Au vu de ce qui a été proposé, les objets Entity qui sont fournis aux composants miroir représentent la réification embarquée du composant juste

avant son exécution. Nous faisons l'hypothèse selon laquelle l'évolution des entités n'entrave pas de modifications sur la structure du code binaire ou que, si c'est le cas, la possibilité d'évolution a été prise en compte et la correspondance est assurée à l'exécution. Cela n'est pas le cas si, par exemple, la valeur d'un attribut est utilisée pour allouer dynamiquement des régions de mémoire physique, ou si les règles de conception liant les valeurs des attributs de deux composants ne sont pas maintenues après que l'un des attributs ait été modifié.

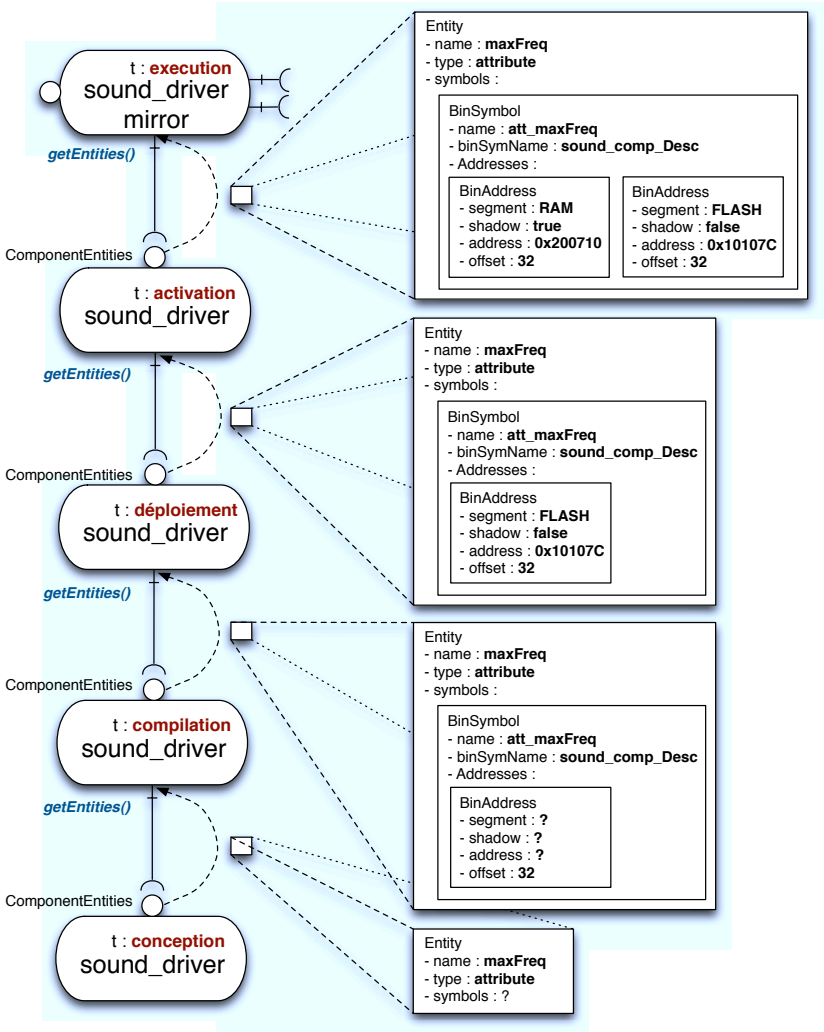


FIGURE 7.5 – Récupération de l'objet Entity correspondant à l'attribut `maxFreq` du composant `sound_driver`.

7.3.2 Lien interne entre miroirs et réifications embarquées

Les composants miroir ont besoin d'établir un lien interne (cf. 5.2.3.3) avec leurs homologues embarqués afin de mener les activités d'évolution dans la plate-forme

embarquée. Ce lien est matérialisé par un ensemble de composants que nous appelons *agents d'évolution*. Ces agents assurent le transfert des actions d'évolution représentées par les interfaces de réflexion des composants miroir depuis l'environnement *off-site* à l'environnement *in-site*, raison pour laquelle ces agents sont distribués dans les deux environnements.

Pour assurer la communication *off-site*/*in-site*, les agents d'évolution reposent sur l'existence d'un moyen de communication entre les deux environnements. Nous n'imposons pas de restriction particulière vis-à-vis la nature de cette communication (moyen physique, protocole utilisé, type de communication) et nous supposons qu'un lien physique est déjà établi et que les agents d'évolution utilisent ce lien.

Le nombre et la nature des agents d'évolution sont susceptibles de varier en fonction des caractéristiques des plates-formes *in-site* et *off-site*. Nous donnons ici une description des interfaces des agents d'évolution, la description sera plus exhaustive lorsque nous déploierons ces agents sur des plates-formes concrètes au chapitre suivant.

La figure 7.6 présente de façon schématique (a) le rôle du lien interne par rapport aux réifications à l'exécution d'un composant et (b) la matérialisation de ce lien en forme d'agents d'évolution.

- `Evolution Front-End` : ce composant offre des interfaces de type `MemoryController` et `EmbeddedLCC` requises par les composants miroir. Après un nombre de traitements spécifiques aux plates-formes d'exécution, il envoie des messages typés à son homologue embarqué.
- `ComDevice` : ces composants encapsulent tout ce qui est relatif aux politiques de communication existant entre les deux environnements *off-site* et *in-site*, représentant le moyen de transmission des messages précédemment évoqués.
- `Evolution Back-End` : ce composant embarqué dans le dispositif est chargé de réceptionner les messages transmis, les interpréter et exécuter les actions d'évolution qu'ils contiennent. Il peut établir des liaisons avec les interfaces de type `LifeCycleController` des réifications embarquées afin de propager les actions relatives à la gestion d'état des composants.
- `MemoryManager` : ce composant encapsule tout ce qui est relatif à la gestion de la mémoire physique dans le dispositif. Il est utilisé par l'`Evolution Back-End` afin de mener les opérations visant la modification des valeurs de la mémoire qui se dérivent des actions d'évolution.

7.3.3 Réifications embarquées

Le traitement d'un bon nombre d'activités liées à l'évolution des entités dans les composants miroir, permet d'appliquer des optimisations sur les réifications embarquées, optimisations qui visent la minimisation de l'impact des IdE sur la performance du logiciel embarqué.

Par rapport aux éléments qui constituent les IdE (cf. 7.1), nous remarquons que les composants miroirs, les agents d'évolution et les composants liés à ceux-ci, prennent en charge les services de réflexion concernant l'entité et les méta-données

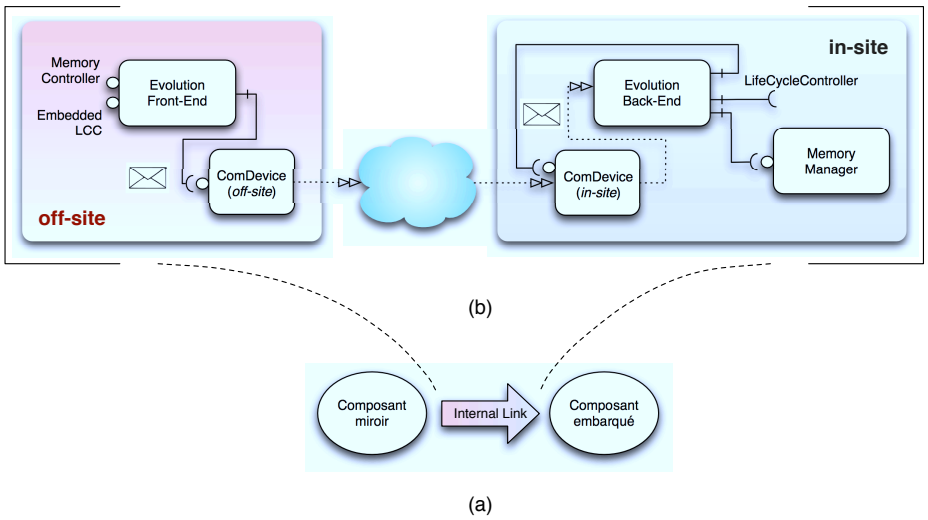


FIGURE 7.6 – Materialisation des liens internes.

associés à ces services. Les réifications embarquées continuent à inclure les métadonnées ainsi que les interfaces de gestion du cycle de vie, dans la mesure ou celles-ci sont étroitement liées au comportement du composant. Le tableau 7.8 présente le niveau d'optimisation appliquée à chaque entité appartenant à cette catégorie d'évolution.

Entité	**Optimisation**
Attributs	A3
Interfaces serveur	IS1
Liaisons	L3
Composites	CO1.2
Composants (mono/multi instance)	P1

TABLE 7.8 – Niveau d'optimisation assigné aux entités dans la catégorie Évolution non prédéfinie.

- *Attributs* : le membre de la structure `attsDesc` qui contient la valeur de l'attribut doit être généré, dans la mesure où c'est l'adresse de celui-ci à qui l'objet Entity fera référence. Cela implique un niveau d'optimisation A3.
- *Liaisons* : les membres des structures `cltItfsDesc` et `srvItfsDesc` faisant référence aux interfaces cliente et serveur qui participent à la liaison, doivent être générés. Un niveau d'optimisation L3 est donc appliqué.
- *Interfaces serveur* : le membre de la structure `srvItfsDesc` qui fait référence à l'interface doit être généré. Cela implique un niveau d'optimisation IS1.
- *Composites* : la composition est une abstraction utilisée à la conception mais réifiée à l'exécution par des nouvelles liaisons et interfaces. L'information sur

l'assemblage interne d'un composite est capturée par le composant miroir et donc sa réification embarquée peut disparaître, ce qui est l'effet du niveau d'optimisation CO1.2.

– *Composants mono/multi instance* : dans la mesure où les politiques d'évolution concernant le nombre d'instances d'un type de composant ne sont connues qu'après le déploiement du logiciel, le type de composant doit être considéré comme multi-instanciable par défaut, ce qui implique un niveau d'optimisation P1.

Les niveaux d'optimisation proposés font partie de ceux qui obtiennent le gains les plus importants pour chaque entité (cf. 6.7), ce qui est en concordance avec l'objectif visé de réduire l'impact dû à l'évolution sur la performance du logiciel.

7.4 Infrastructures d'évolution au niveau système

L'infrastructure d'évolution d'un système est composée des infrastructures d'évolution qui sont générées pour chaque entité en fonction de la catégorie d'évolution assignée à celle-ci lors de l'étape de conception. Cela nous permet de générer des IdE « sur mesure » en fonction des besoins exprimés lors de la phase de conception du système.

Tout au long de ce chapitre nous avons considéré que les catégories d'évolution peuvent être assignées de manière indépendante aux entités d'un composant. Bien que cela soit vrai pour la plupart des cas, un nombre limité d'effets de bords et de conflits entre niveaux d'optimisation doivent être pris en compte.

- Pour une interface serveur catégorisée Non Évolution, le niveau d'optimisation IS3 est appliqué seulement si aucune liaison n'est dirigée vers cette interface ou si toutes les liaisons qui s'y dirigent sont marquées *inline*.
- Pour une interface serveur catégorisée Non Évolution, le niveau d'optimisation IS2 est appliqué seulement si toutes les liaisons dirigées vers cette interface serveur sont elles aussi catégorisées Non Évolution. Dans le cas contraire l'interface est ré-catégorisée Évolution Dynamique et le niveau IS0 est appliqué. En effet, si l'une des liaisons est plausible d'évoluer, le composant client associé offrira une interface de type `BindingController`. Dans l'invocation de la méthode `bind` de celle-ci, il est nécessaire de fournir un identifiant de l'interface serveur `serverItfId` qui est obtenu par l'invocation de la méthode `getInterface` de l'interface `ComponentIdentity` du composant qui offre l'interface serveur. Le niveau d'optimisation IS0 est le seul qui implique l'ajout de l'interface `ComponentIdentity`.
- Un composant composite classé Évolution Dynamique impose à ses sous-composants (i) que toutes les interfaces serveur qui font objet de délégation de services du composite soient classées Évolution Dynamique ou Évolution non prédéfinie, et (ii) que les liaisons clientes générées dans la réification du composite soient marquées Évolution Dynamique. Cela est du au fait que, afin de modifier l'assemblage interne du composite, celui-ci doit offrir l'interface `BindingController`et les sous-composants l'interface `ComponentIdentity`.

- Une liaison classée Évolution non prédéfinie impose ce même classement à l'interface serveur vers qui elle est dirigée. Cela puisque l'adresse du membre de la structure `srvItfsDesc` qui fait référence à l'interface serveur est nécessaire à l'objet Entity associé à la liaison.

D'autre part, les composites auxquels on assigne les catégories Évolution Statique et Évolution non prédéfinie, disparaissent suite au niveau d'optimisation CO1.2. Dans ce cas :

- Les interfaces serveur des sous-composants auxquelles les services du composite sont délégués, héritent la catégorie d'évolution des interfaces serveur correspondantes du composite.
- Les liaisons à partir des sous-composants et qui sont dirigées vers des composants à l'extérieur du composite, héritent la catégorie d'évolution des liaisons à partir du composite correspondant.

7.5 Synthèse

Nous nous sommes intéressé dans ce chapitre aux informations qui décrivent le "comment" de l'évolution à l'exécution, et plus concrètement sur la relation entre les Infrastructures d'Évolution (IdE) qui rendent possible les activités d'évolution, et l'impact de celles-ci sur la richesse d'évolution et sur la performance du système. Nous avons décrit le "comment" de l'évolution à partir d'un ensemble de réponses aux questions suivantes :

- l'entité évoluera après son déploiement ?
- quelles sont les politiques d'évolution de l'entité ? Quelles sont les "valeurs" que l'entité peut prendre après son déploiement ?
- à quel instant précis de l'exécution l'entité évoluera ?
- qui déclenche les actions d'évolution ? le composant lui-même ? un intergiciel dans le dispositif ? un agent externe au dispositif ?

Enfin, une dimension qui est transversale à ces questions :

- à quel moment du cycle de vie logiciel cette information est disponible ?

En fonction de ces dimensions nous avons défini 4 **catégories d'évolution** qui peuvent être assignées à chaque entité du modèle à composants : attribut, interface serveur, interface cliente, liaison et composants composites et primitifs. L'une de ces catégories ("Pas d'évolution") correspond au cas trivial où l'on sait à la conception que l'entité n'évoluera jamais après son déploiement. Les autres catégories ("Évolution dynamique", "Évolution statique" et "Évolution non prédéfinie") se différencient entre elles par rapport aux réponses que leurs IdE fournissent aux questions posées précédemment et aux optimisations que l'on peut effectuer au moment de la construction de la réification à l'exécution de l'entité.

Le tableau 7.9 présente la façon dont les catégories proposées se relient aux question posées. Selon quand la réponse et quel réponse est disponible, nous proposons les catégories d'évolution les plus adaptées. Lorsqu'on connaît à la conception

les politiques d'évolution de l'entité, nous lui assignons de préférence la catégorie "Évolution statique", même si la catégorie "Évolution dynamique" est aussi valide. Dans certains cas où plusieurs catégories sont adaptées, le choix est fait en fonction d'autres aspects extra-fonctionnels, parmi lesquels on trouve l'impact des IdE sur la performance du logiciel.

Dimensions du "comment"	*L'information est disponible ?*		
	Oui *Quand cette info est-elle disponible ?*		Non
	Conception	Post-déploiement	
L'entité évoluera après son déploiement ?			
Oui	statique, dyn, nonDef	dyn	nonDef
Non	nulle	nonDef	nonDef
Quelles valeurs l'entité peut prendre ?	statique	nonDef, dyn	nonDef, dyn
A quel moment de l'exécution l'entité évoluera ?	statique	nonDef, dyn	nonDef, dyn
Qui déclenche les actions d'évolution ?			
Composant lui-même	dyn, statique, nonDef	-	dyn
Intergiciel	dyn, statique	dyn	dyn
Agent externe	nonDef, dyn	nonDef, dyn	nonDes, dyn

TABLE 7.9 – Dimensions du "comment" de l'évolution et catégories d'évolution proposées.

Maintenant, nous présentons une synthèse des principales caractéristiques des catégories d'évolution :

- **Évolution Dynamique**. Cette catégorie correspond au cas par défaut de la plate-forme Fractal/Think. Les IdE qui sont générées couvrent quasiment toutes les dimensions du "comment" de l'évolution et ont une richesse d'évolution forte. Néanmoins, les niveaux d'optimisation des entités (cf. tableau 7.3) nécessaires à leur mise en œuvre sont faibles, ce qui nous laisse penser que le coût à payer en termes d'impact sur la performance du logiciel est non négligeable. Ces propos seront confirmés au chapitre suivant sur l'évaluation de l'approche.
- **Évolution Statique**. Cette catégorie se base sur la connaissance détaillée, au moment de la conception du système, de la manière dont l'entité évoluera et du moment à l'exécution où elle le fera (i.e. des politiques d'évolution de l'entité). La richesse d'évolution se voit donc limitée aux actions d'évolution définies à la conception, ce qui rend cette catégorie non adaptée aux évolutions imprévues. Les niveaux d'optimisation des entités (cf. tableau 7.5) laissent voir que l'impact sur la performance du logiciel est minimal. Néanmoins, si l'on veut mettre en œuvre de façon optimale cette catégorie d'évolution, il est nécessaire de disposer d'outils d'analyse statique du comportement des

composants. Ces outils ne sont pas à ce jour disponibles dans Fractal/Think.

- **Évolution non prédéfinie**. Cette catégorie couvre une bonne partie des dimensions qui sont couvertes par la catégorie Évolution dynamique, elle impose peu de restrictions par rapport aux actions d'évolution qui peuvent être exécutées, et elle est spécialement adaptée aux évolution imprévues et à la commande à distance des actions d'évolution. La haute richesse d'évolution et les niveaux d'optimisation des entités (cf. tableau 7.8) nous laissent penser que sa mise en œuvre propose un compromis entre richesse d'évolution et performance du logiciel, évoqué tout au long de cette thèse.

  Cette mise en œuvre consiste en un redéploiement d'une partie de l'IdE, via :
  - les composants miroir,
  - les liens verticaux entre réifications de composants, pour récupérer des informations sur la manière dont les entités embarquées ont été compilées et déployées,
  - les liens internes entre réifications des composants à l'exécution, pour mener à bien les action d'évolution.

Enfin, la génération indépendante d'IdE pour chacune des instances des entités du modèle à composants présentes dans un système, nous permet d'assigner des catégories d'évolution à chaque entité. Ainsi, nous sommes en mesure d'adapter de manière optimale la génération de code en fonction des besoins d'évolution du système.

Chapitre 8

Evaluation

"In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering" - Donald Knuth [Knuth 1979]

Sommaire

8.1 Plate-forme d'évaluation 141
8.1.1 Motivation et choix 141
8.1.2 Architecture matérielle 143
8.1.3 Logiciel embarqué 145
8.2 Optimisation des composants logiciel 147
8.2.1 Propriétés d'optimisation à la conception 147
8.2.2 Comparaison avec le code *legacy* sur le *bare-metal* 147
8.2.3 Cas d'étude : développement de pilotes de périphériques . . . 151
8.2.4 Cas d'étude : un RTOS à base de composants logiciel 155
8.2.5 Conclusion sur l'évaluation des optimisations 158
8.3 Infrastructures d'évolution 158
8.3.1 Etude de l'impact des Infrastructures d'évolution 159
8.3.2 Infrastructures d'évolution non prédéfinies 161
8.3.3 Conclusion sur l'évaluation des IdE 173

Dans ce chapitre nous évaluons les contributions de cette thèse. Dans un premier temps nous présentons les propriétés souhaitées pour la plate-forme d'exécution utilisée dans nos évaluations, et présentons les caractéristiques matérielles et logicielles de la plate-forme choisie, au chapitre 8.1. Les chapitres 8.2 et 8.3 sont consacrés à l'évaluation des techniques d'optimisation et aux infrastructures d'évolution (IdE) proposées aux chapitres 6 et 7.

8.1 Plate-forme d'évaluation

8.1.1 Motivation et choix

Nous avons identifié un ensemble de caractéristiques que la plate-forme d'évaluation doit posséder, en plus de celles évoquées au chapitre 2 de cette thèse, et notamment l'appartenance à la famille des microcontrôleurs. Dans un premier temps, et comme guide général, nous souhaitions qu'elle soit suffisamment générique afin de pouvoir reproduire des cas d'utilisation rencontrés dans les systèmes embarqués et ainsi capturer le plus possible l'hétérogénéité propre aux applications du domaine. Ensuite, des lignes directrices plus précises ont été identifiées :

- La quête de la généralité ne doit pas se faire au détriment des aspects économiques qui, même s'ils ne sont pas traités au cours de cette thèse, participent au choix des plates-formes matérielles et logicielles. Plus concrètement, concernant l'aspect matériel, la plate-forme choisie doit se trouver dans une gamme intermédiaire concernant la puissance de calcul, les périphériques existants et la mémoire physique disponible.
- La plate-forme choisie doit pouvoir exécuter une grande diversité d'applications de préférence interactives, i.e. impliquant que des acteurs externes à la plate-forme jouent un rôle (ne fût-ce que d'observateurs) dans l'application. Cela signifie l'existence d'une variété des capteurs et d'actionneurs qui puissent être liés à la plate-forme, ce qui à son tour exige de celle-ci des diverses familles de périphériques d'entrée/sortie.
- Lié au point précèdent, nous exigeons de la plate-forme un lien de communication sans-fil, afin de reproduire les applications propres aux systèmes pervasifs ou ubiquitaires [Weiser 1993], ainsi que les réseaux de capteurs.
- Concernant la relation entre les couches matérielles et logicielles, nous demandons qu'elles soient les plus indépendantes l'une de l'autre. Cela implique notamment que la plate-forme matérielle ne doit pas être couplée avec une couche logicielle, un compilateur ou un langage de programmation quelconque. Par ailleurs, il doit être possible de déployer une application directement sur la plate-forme matérielle sans passer par une couche logicielle sous-jacente. De même, l'exécution de la plate-forme logicielle, si elle existe, ne doit pas dépendre de l'existence d'une fonctionnalité matérielle spécifique (e.g. une MMU).
- Enfin, en ce qui concerne l'outillage requis par le développement logiciel (langages de programmation, compilateurs, IDE), il doit être également générique et en concordance avec les pratiques industrielles courantes.

Nous avons trouvé la plate-forme **Lego Mindstorms NXT** [Lego 2010] idéale pour évaluer nos contributions. Elle est composée d'une brique centrale de traitement (que nous appelons *NXT*) à laquelle on peut connecter jusqu'à 4 capteurs parmi des capteurs de :

- contact
- lumière (noir ou blanc)
- niveaux de son entre 3 et 6 KHz
- proximité (0 à 20 cm), via un capteur ultrason
- boussole
- gyroscope
- accélération

La brique NXT offre également la connexion possible de 3 servomoteurs, un écran LCD cd 100x64 pixels, un émetteur de son et 4 boutons. Au niveau de la communication, un port USB est offert ainsi que des capacités de communication sans-fil Bluetooth.

La plate-forme Lego Mindstorms a été originalement destinée aux jeunes comme une plate-forme d'initiation à la robotique. Néanmoins, derrière ce visage ludique se cache un microcontrôleur Atmel [Atmel 2010c] AT91SAM7S256 [Atmel 2010b]

de la famille ARM7TDMI [ARM 2004]. Ce microcontrôleur RISC 32 bits embarque 256 KB de mémoire FLASH et 64 KB de mémoire RAM, ainsi qu'une variété de périphériques qui seront évoqués au chapitre 8.1.2. Cette famille de processeurs a connu un grand succès depuis son introduction, elle est embarqué dans des produits comme la console de jeux Nintendo DS, l'iPod, certains des téléphones Nokia ou le contrôleur *Home Automation* BBC-SD de *American Auto-Matrix*.

Coté logiciel, Lego fournit une chaîne de développement à très haut-niveau d'abstraction qui repose sur un système d'exploitation *ad-hoc* avec une politique d'ordonnancement de type *round-robin*. Le firmware publié par Lego est lié au compilateur IAR [IAR 2010]. Néanmoins, des nombreux projets [Pedersen 2007, NXTGCC 2010, LejOS 2010a, NXTOSEK 2010, nxOS 2010] fournissent des firmwares en forme de code source C qui peut être compilé par GCC. Nous nous sommes basés sur l'un d'entre eux, nxOS [nxOS 2010], pour développer une couche d'abstraction du matériel (*Hardware Abstraction Layer*, HAL) et les pilotes de périphériques. Contrairement à la partie matérielle, où nous sommes partis sur une plate-forme existante, en ce qui concerne le logiciel nous avons développé et adapté du code source existant. Nous détaillons ce point au chapitre 8.1.3.

FIGURE 8.1 – Plate-forme Lego Mindstorms NXT

8.1.2 Architecture matérielle

La figure 8.2 présente rapidement les principaux composants matériels de la brique NXT, ainsi que les protocoles de communication utilisés pour les lier entre elles.

La communication entre le processeur principal AT91SAM7S256 et les périphériques externes se fait via un coprocesseur à 8 bits ATMega48 [AVR 2010a] de la famille AVR [AVR 2010b], lié au premier par un bus série TWI (*Two Wire Interface*), sauf pour le capteur de proximité et l'acquisition des révolutions des servomoteurs, qui sont liés directement au processeur principal par un bus série synchrone I2C

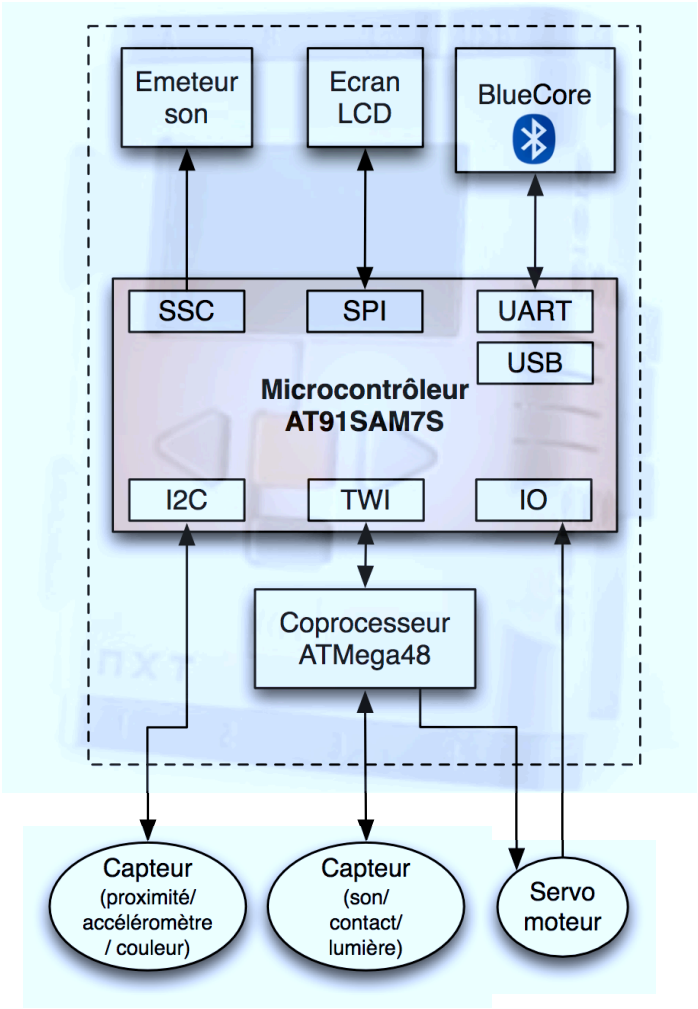


FIGURE 8.2 – Architecture matérielle de la brique NXT

(*Inter Integrated Circuit*) et sur un des ports d'entrée/sortie, respectivement [1]. Le processeur principal obtient de l'AVR la valeur des capteurs et l'état des boutons, et envoie à celui-ci la fréquence du PWM de l'AVR utilisé dans la lecture des servomoteurs, la vitesse et le mode de frein souhaités pour ceux-ci. A cause des contraintes liées à l'AVR, le lien entre les deux processeurs doit être assuré périodiquement, au minimum toutes les 10 ms.

La communication entre le processeur ARM7 et l'écran est assurée via un bus via SPI (*Serial Peripheral Interface*). La communication entre le processeur et l'émetteur de son est assurée via un bus SSC (*Serial Synchronous Controller*). La brique contient également une puce BlueCore [CSR 2010] qui gère les couches basses de communication Bluetooth, et qui est contrôlée par le processeur principal via un bus UART (*Universal Asynchronous Receiver Transmitter*).

8.1.3 Logiciel embarqué

La plate-forme permet la ré-écriture de la mémoire du microcontrôleur ARM7 via son port USB, ce qui nous donne la possibilité de remplacer le logiciel qui s'exécute sur le processeur principal. Afin de remplir le besoin de pilotes logiciels pour les périphériques et celui d'un exécutif, nous avons conçu des composants basés sur le code source C fournit par le projet NXOS [nxOS 2010] pour les premiers (cf. 8.1.3.1), et sur le système d'exploitation temps-réel μC/OS-II [Micrium 2010, Labrosse 2002] pour le deuxième (cf. 8.1.3.2).

L'avantage de cette configuration par rapport à l'existant est que tout le code source C est disponible, qu'il peut être compilé par le compilateur GCC, très répandu dans l'industrie, et qu'il peut être édité en utilisant des IDE courants comme Eclipse. L'approche composant nous permet également de concevoir des systèmes sur mesure par rapport aux besoins d'évaluation.

8.1.3.1 HAL et pilotes de périphériques

Le projet NXOS [nxOS 2010] fournit des pilotes basiques pour une bonne partie des capteurs, les servomoteurs et les autres périphériques externes (LCD, émetteur de son). Il fournit également des pilotes USB et Bluetooth, ainsi que des pilotes pour les périphériques de bas niveau du microcontrôleur (gestion d'interruptions, compteurs, convertisseurs analogue/numérique) et du code gérant la communication avec l'AVR. Une avantage de NXOSpar rapport à d'autres projets est que leur code n'est pas lié à une plate-forme logicielle sous-jacente (e.g. un exécutif) et il peut s'exécuter sur ce qui est appelé le niveau *bare-metal*.

Nous avons créé une bibliothèque de composants Fractal/Think basés sur NXOS. Des fichiers de description de composants (ADL) et d'interfaces (IDL) ont été crées dans ce but. Le code source C a été adapté à la plate-forme de développement Fractal/Think en insérant des annotations en forme de commentaires C [Think Team 2009], comme illustré au chapitre 6 de cette thèse. Dans le temps, la bibliothèque de composants s'est vue enrichie par des modifications au niveau des

1. Le bus I2C est également utilisé pour lier la brique NXT et les capteurs non inclus par défaut : accéléromètre, boussole, couleur (rouge/vert/bleu).

assemblages et du contenu des composants. En particulier, nous avons mené une réflexion sur la génération optimisé des pilotes de périphériques (cf. 8.2.3).

8.1.3.2 Executif temps-réel

μC/OS-II [Micrium 2010, Labrosse 2002] est un noyau de système d'exploitation temps réel (RTOS) multitâche et préemptif. Il est implémenté en ANSI C et certifié par la *U.S. Federal Aviation Administration* (FAA) dans le cadre de son utilisation dans des équipements avioniques. Il a été massivement utilisé dans de nombreux systèmes critiques embarqués et porté sur des très nombreuses architectures matérielles : ARM7/9, AVR32, Cortex-M3, PIC32/24, PowerPC, MSP430, Coldfire, TriCore, 80x86, MIPS R3000/4000, entre autres.

μC/OS-II peut gérer jusqu'à 64 tâches. μC/OS-II comprend des sémaphores, des *flags* d'évènements, des *mutex*, des boîtes à lettres et des files d'attente de messages, ainsi qu'un gestionnaire de blocs de mémoire à taille fixe. Le temps d'exécution pour la plupart des services fournis par μC/OS-II est à la fois constant et déterministe, il ne dépend pas du nombre de tâches en cours d'exécution. Cette propriété est une exigence obligatoire pour les systèmes temps réel afin d'éviter le phénomène connu comme le *kernel jitter* [Burns 1995]. Pour une description plus détaillée de μC/OS-II nous orientons le lecteur vers [Labrosse 2002].

Dans [Loiret 2009] nous avons componentifié μC/OS-II sur le modèle Fractal/-Think, puis nous avons généré une implémentation optimisée. Le chapitre 8.2.4 traite les aspects soulevés par cette démarche.

Conclusion sur la plate-forme d'évaluation. Les caractéristiques de la plate-forme matérielle Lego Mindstorms NXT, avec les bibliothèques de composants logiciels développées et qui se basent sur du code source ANSI C existant, est en plein accord avec les lignes directrices tracées au début du chapitre. En effet :

- Les ressources physiques disponibles sur le microcontrôleur (32 bits, 256 KB de mémoire FLASH, 64 KB de mémoire RAM, jusqu'à 55 MHz et 0.9 MIPS/Mhz) se trouvent dans une gamme intermédiaire entre les plate-formes orientées vers le traitement d'audio ou video (e.g. 1 MB de RAM) et celles orientées vers les réseaux de capteurs (e.g. $<$ 10 KB de RAM).
- La grande variété de périphériques disponibles permet de reproduire des applications très hétérogènes, même si un accent sur des applications orientées robotique est toujours présent. L'existence d'un lien Bluetooth permet également de déployer des applications typiques des milieux pervasifs.
- L'approche composant appliqué à toutes les couches logiciel (applicatif, exécutif et pilotes de périphériques) nous permet de mieux identifier les dépendances entre celles-ci et de déployer des systèmes à la mesure de nos besoins d'évaluation. De plus, l'outillage utilisé pour la conception et le développement est celui employé dans des nombreux projet industriels : Eclipse IDE pour la conception, `gcc` pour la compilation, `GNU ld` pour la résolution des liens binaires.

8.2 Optimisation des composants logiciel

Dans ce chapitre, nous évaluons les techniques d'optimisation des réifications embarquées des composants proposées au chapitre 6 de cette thèse. Nous définissons d'abord des propriétés de haut-niveau, applicables à chaque entité du modèle à base de composants, qui indiquent à la chaîne de compilation les techniques d'optimisation à appliquer lors de la construction de la réification de l'entité à la compilation (cf. 8.2.1). Ensuite, nous comparons la performance du logiciel conçu en suivant une méthodologie classique, centrée sur le code source (élément de référence), avec une méthodologie basée composants logiciel, avec ou sans optimisation (cf. 8.2.2). Enfin, nous décrivons deux cas d'application de nos optimisation sur le développement de pilotes de périphériques (cf. 8.2.3) et de systèmes d'exploitation temps réel (cf. 8.2.4).

8.2.1 Propriétés d'optimisation à la conception

La plate-forme Fractal/Think permet l'extension de la chaîne de compilation afin de prendre en compte des *propriétés* [Anne 2009] définies par l'utilisateur, applicables aux entités du modèle : composants, contenu, attributs, liaisons, interfaces serveur et interfaces clientes. En termes pratiques, les *propriétés* sont ajoutées à des points précis des fichiers ADL et elles ont la forme [Think Team 2009] :

$$[< nomProprieté >=< valeur >]$$

Donnant suite aux travaux effectués dans [Lobry 2008], nous avons défini dans [Lobry 2009] un ensemble de *propriétés* qui correspondent à certains des niveaux d'optimisation présentés au chapitre 6 et aux techniques classiques synthétisées au chapitre 3.7. La correspondance entre ces techniques classiques et les niveaux d'optimisation définis a été présentée dans le tableau 6.15.

Le tableau 8.1 [2] présente les *propriétés* Fractal/Think que nous avons définis. On y retrouve l'entité pour laquelle la propriété est définie, les valeurs que l'on peut assigner à la propriété, et le niveau d'optimisation qui correspond à ces valeurs. Les *propriétés* marquées avec une étoile (*) sont celles appliquées par défaut par la chaîne de compilation étendue.

8.2.2 Comparaison avec le code *legacy* sur le *bare-metal*

Notre premier objectif en ce qui concerne les optimisations des réifications des composants a été de valider l'efficacité des techniques proposées. Pour cela nous avons reproduit un scénario où un même logiciel est conçu se basant sur deux paradigmes différents de développement : l'un centrée sur le code source et l'autre basé sur des composants logiciel Fractal/Think. Ensuite, nous devions comparer la performance à l'exécution associée à chaque scénario afin de mesurer l'impact de

2. Dans ce tableau, pour la propriété `[single=true]` concernant les composants, le niveau P2.2 est appliqué si la propriété `[single=true]` est appliquée aux interfaces serveur du composant et si ces interfaces sont effectivement les seules de ce type.

Entité	Propriété	Niveau d'optimisation
Attribut	`[const=true]`	A4
	`[const=false]*`	A3
Interface serveur	`[garbage=true]`	IS3
	`[garbage=false]*`	IS1
	`[single=true]`	-
Liaison	`[static=true]`	L4
	`[static=inline]`	L5
	`[static=false]*`	L3
Composite	`[bag=true]`	CO1.2
	`[bag=false]*`	CO1.1
Primitif	`[single=true]`	P2.1, P2.2
	`[single=false]*`	P1

TABLE 8.1 – Propriétés d'optimisation proposées

l'approche composant sur celle-ci. Comme critères d'évaluation nous avons choisi l'occupation mémoire et le temps d'exécution des méthodes des interfaces (ou des fonctions pour le scénario centré sur le code source).

Nous avons donc comparé l'exécution d'un sous-ensemble de la base de code originale de NXOS, développée par David Anderson (Google, Inc.), Jérôme Flesh (NetAsq) et Maxime Petazzoni (Montavista Software, Inc.), entre autres [Petazzoni 2009], avec une application équivalente basée sur une des premières versions de la bibliothèque de composants développée dans le cadre de cette thèse. Au-dessus de cette couche logicielle, nous avons développé une simple application `demo` exécutant une unique et simple tâche.

La figure 8.3 presente une architecture simplifiée du logiciel étudié. Il est composé de 7 composants primitifs : `lcd`, `avr`, `twi` et `lowlevel` qui constituent une première couche d'abstraction du matériel ; `display` et `sound` sont des pilotes de l'écran et de l'émetteur de son, respectivement ; `demo` est le composant applicatif.

8.2.2.1 Occupation mémoire

Nous avons évalué l'occupation mémoire pour 5 scénarios qui sont décrits par le tableau 8.2. Le tableau 8.3 présente les résultats obtenus par rapport au code (TEXT) et aux données (DATA) binaires [3]. Puisque nous nous intéressons tout particulièrement à l'occupation mémoire, nous forçons le compilateur de bas niveau GCC à optimiser agressivement cet aspect, en activant le flag de compilation `-Os`.

Les résultats obtenus montrent d'abord que l'utilisation de l'approche composants Fractal/Think a un impact sur l'occupation mémoire du logiciel de +15,6%, mais que l'application des optimisations les plus agressives parmi celles proposées au chapitre 6 réussissent à rendre nul cet impact. Les résultats des scénarios d'optimisation partielle illustrent la possibilité d'appliquer des niveaux distinctes d'optimisa-

3. Cette distinction sera faite tout au long de ce chapitre. Nous ne faisons aucune supposition sur le type de mémoire physique sur laquelle le code et les données seront déployés, même si classiquement le premier est stocké dans une mémoire ROM ou FLASH, et les deuxièmes sur une mémoire RAM après une procédure de *shadowing* à partir des adresses de la ROM/FLASH.

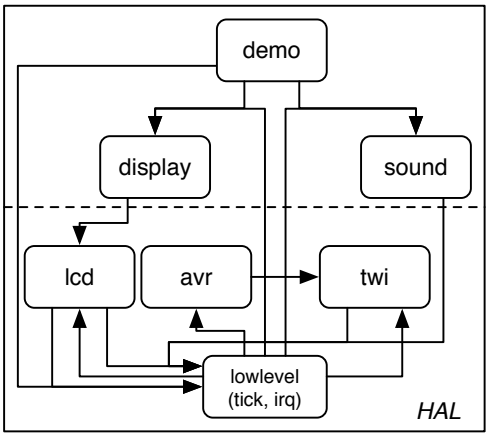


FIGURE 8.3 – Architecture simplifiée du logiciel

tion sur les entités du système. Notamment, pour le scénario 5, la non-optimisation de composants qui représentent près de 30% de l'occupation mémoire du logiciel, produit un surcoût de seulement 2,35%. Les surcoûts des optimisation partielles d'un système sont étroitement liés au nombre de composants non-optimisés et à leur granularité.

8.2.2.2 Temps d'exécution des méthodes

Afin d'évaluer le temps d'exécution des méthodes, nous avons mené un exercice similaire à celui mené pour la base de code NXOS, cette fois sur une application démo d'un émulateur de la plate-forme MCBSTM32 [Keil 2010] qui embarque un microcontrôleur STM32F103RB [Keil 2009] (32 bits, 72 MHz, 128 KB Flash, 20 KB SRAM) de la famille Cortex-M3 [4][ARM 2006]. En effet, si nous avons basé l'essentiel de nos mesures sur la plate-forme Lego Mindstorms NXT, ces estimations illustrent que les résultats ne sont pas liés à une plate-forme d'exécution (microcontrôleur) particulière.

L'application contient deux composants, l'un qui gère le *tick* matériel et qui collecte des données via un ADC, et l'autre qui fournit des services pour imprimer des données sur un LCD.

Le tableau 8.4 présente le temps moyen d'exécution (en μs) des 4 méthodes de l'interface serveur du composant qui gère l'écran LCD, pour 3 scénarios équivalents aux 3 premiers évoqués précédemment.

Les résultats obtenus confirment, sur ce critère de performance, l'efficacité des optimisations agressives, où l'on obtient un surcoût nul avec l'utilisation d'une approche composant. Les surcoûts des méthodes sous le scénario 1 sont étroitement liés au comportement encapsulé par leur définition, et notamment au nombre et fréquence d'accès aux attributs, d'appels à des méthodes privées ou d'appels à des

4. N'ayant pas trouvé un émulateur du microcontrôleur AT91SAM7S qui satisfasse nos attentes, nous nous sommes tournés vers cette plate-forme à cause des ressemblances entre les deux.

Scénario	Description
1. Référence	Base de code nxOS, approche non basé composants.
2. Fractal/Think par défaut	Les optimisations non surlignées du tableau 8.1 sont appliquées. Cela correspond au comportement par défaut du compilateur Fractal/Think : les méta-données sur les entités du modèle sont générées et les interfaces orientées réflexion ne sont pas ajoutées par défaut, elles doivent être explicitement ajoutées par le développeur.
3. Optimisé	Les optimisations les plus agressives, celles surlignées dans le tableau 8.1, sont appliquées sur tous les composants.
4. Partialement optimisé (A)	Tous les composants sauf `demo` sont optimisés comme dans le scénario 3.
5. Partialement optimisé (B)	Seulement les composants HAL (`lcd`, `avr`, `lowlevel` et `twi`) sont optimisés comme pour le scénario 3.

Table 8.2 – Description des scénarios d'évaluation

Scénario	TEXT	DATA	TOTAL
1. Référence	8602	910	9512
2. Default	9766	1230	10996
surcoût	*13,53%*	*35,16%*	*15,60%*
3. Optimisé	8602	910	9512
surcoût	*0,00%*	*0,00%*	*0,00%*
4. Partialement optimisé (A)	8670	1006	9676
surcoût	*0,79%*	*10,55%*	*1,72%*
5. Partialement optimisé (B)	8698	1038	9736
surcoût	*1,12%*	*14,07%*	*2,35%*

Table 8.3 – Occupation mémoire des scénarios d'évaluation

méthodes clientes, puisque ces sont les indirections générées par la chaîne de compilation qui sont à l'origine des surcoûts.

Scénario	nom de la méthode			
	init	**clear**	**print**	**bargraphXY**
1. Reference	11,208	0,264	2,417	3,226
2. Défaut	12,111	0,292	2,736	3,559
surcoût	*8,06%*	*10,61%*	*13,21%*	*10,32%*
3. Optimisé	11,208	0,264	2,417	3,226
surcoût	*0,00%*	*0,00%*	*0,00%*	*0,00%*

Table 8.4 – Temps d'exécution (μs) des méthodes de l'interface LCD

Conclusion. Les optimisations agressives proposées au chapitre 6 et mises en œuvre via les propriétés de l'ADL Fractal/Think (cf. tableau 8.1) permettent la mise en place d'une approche de développement basée sur des composants logiciels sans perte de performance par rapport aux approches basées sur le code source.

Les expériences décrites ici ont illustré cet aspect par rapport aux deux critères d'optimisation évalués : l'occupation mémoire et le temps d'exécution.

Les chapitres suivants montrent comment ces propos restent vrais lorsqu'on s'intéresse à des logiciels plus complexes comme des pilotes de périphériques (cf. 8.2.3) ou des systèmes d'exploitation (cf. 8.2.4), dont le processus de développement prend compte des particularités propres à ces types de logiciels.

8.2.3 Cas d'étude : développement de pilotes de périphériques

Le développement de logiciel de bas niveau, et notamment celui de pilotes de périphériques, doit faire face à un ensemble de préoccupations très spécifiques au domaine, telles que :

- Les particularités techniques des plate-formes matérielles : adresses de registres de configuration ou de lecture/écriture, politiques d'interruptions, gestion de compteurs matériels,
- Les politiques de stockage d'information, dans la mesure où un pilotes agit comme un *buffer* de données entre l'application et les ressources gérées par celui-ci,
- La tolérance aux pannes et l'administration autonome du service. En particulier, la panne d'une application ne doit pas affecter l'accès à une ressource partagée avec d'autres applications. Le pilote doit pouvoir isoler la panne ou se réinitialiser de façon autonome,
- La portabilité entre plates-formes logicielles et matérielles, à la source de standards d'API comme POSIX.

Dans [Navas 2009a] nous proposons un ensemble de stratégies de conception de pilotes de périphériques, basées sur l'utilisation de modèles à base de composants :

i. Le découplage entre l'application et le pilote du périphérique

ii. L'abstraction des détails du matériel (registres de configuration des processeurs, politiques d'interruption)

iii. L'utilisation extensive de la composition pour gérer la complexité du logiciel

iv. La définition exhaustive des frontières des composants (e.g. via les interfaces serveur/clientes)

v. La conception d'architectures par couches [DeAntoni 2005] pour séparer les activités de capture, de raisonnement et de communication au sein des pilotes.

Il est à noter que les travaux précédents sur la mise en place de stratégies de conception de haut niveau [Gamma 1995, Crnkovic 2004, Angelov 2005, Gay 2005] ne prennent pas en compte l'impact que ce genre de stratégies peut avoir sur la performance du logiciel. Par l'application des techniques d'optimisation évoquées au long de cette thèse, nous arrivons à réduire l'impact que l'adoption de ces stratégies de conception peut avoir sur l'exécution du logiciel, et plus concrètement sur l'occupation mémoire.

Nous avons mis en œuvre et testé l'efficacité de ces stratégies dans le développement de deux pilotes de périphériques de la plate-forme NXT : la gestion de la

communication entre le processeur ARM7 AT91SAM7S et le coprocesseur AVR ATMega48 (cf. 8.2.3.1) et la gestion de l'écran LCD (cf. 8.2.3.2). Aux chapitres suivants nous présentons les scénarios proposées et les résultats obtenus.

8.2.3.1 Pilote du coprocesseur AVR

Comme évoqué précédemment, le processeur principal de la brique NXT envoie et reçoit périodiquement au coprocesseur AVR des informations concernent les moteurs et les capteurs, respectivement. Un bus TWI est utilisé à cette fin. Le communication entre les deux processeurs est considérée critique, dans la mesure où une interruption peut entraîner l'arrêt de tout le système.

Nous avons évalué 4 scénarios de conception basés sur les stratégies présentées précédemment :

1. Isolation du pilote : la gestion du coprocesseur a été isolée du reste du système. Un composant offre des services de rafraîchissement (`update`) et de requête (`query`) des valeurs de capteurs, masquant les détails de communication.
2. Abstraction des dépendances sur le matériel : nous dissocions les aspects logiciel et matériel (dépendant de l'architecture) au niveau des composants.
3. Dépendances vers le matériel orientées services : un composant agnostique aux caractéristiques du microcontrôleur ARM7 a été conçu, il encapsule les aspects du bus TWI qui sont liés au matériel.
4. Couches fines : nous proposons une structure à base de couches pour la gestion du coprocesseur, comme le montre la figure 8.4.

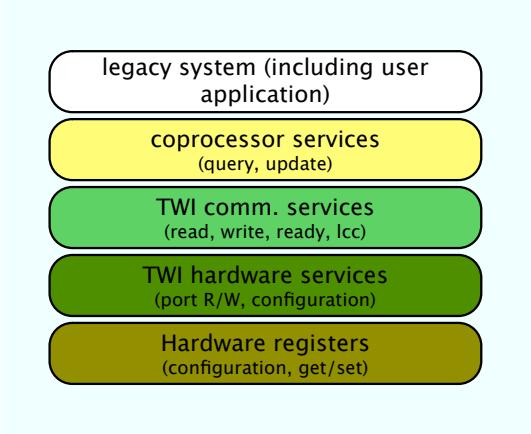


FIGURE 8.4 – Structure à base de couches du pilote du coprocesseur

Nous avons mesuré l'occupation mémoire d'une application minimale non componentifiée (similaire à celle de référence du chapitre 8.2.2) sous les 4 scénarios proposés. Pour chaque scénario, deux mesures ont été faites en fonction des optimisations appliquées sur la partie basée composants (le périphérique) : l'une équivalente

au cas 2 du tableau 8.2, représentant le comportement par défaut de la chaîne Fractal/Think [5], et l'autre représentant le cas 3 de la même table, où des optimisations agressives sont appliquées. Les résultats sont présentés dans le tableau 8.5.

Scénario		TEXT	DATA	TOTAL	Surcoût
Référence		8060	2530	10590	0,00%
1	Défaut	8284	2588	10872	2,66%
	Optimisé	8136	2546	10682	0,87%
2	Défaut	9418	2788	12206	15,26%
	Optimisé	8272	2546	10818	1,78%
3	Défaut	9354	2758	12112	14,37%
	Optimisé	8232	2546	10778	1,78%
4	Défaut	9894	2838	12732	20,23%
	Optimisé	8562	2596	11158	5,36%

TABLE 8.5 – Occupation mémoire (Bytes) du logiciel sous les scénarios proposés - cas coprocesseur

8.2.3.2 Pilote de l'écran LCD

Un exercice similaire a été mené sur le pilote de l'écran du NXT, qui embarque un contrôleur dédié UC1601 [UltraChip 2010]. L'interface avec le processeur ARM7 se fait à travers d'un bus SPI.

Comme pour le cas du pilote du coprocesseur, nous avons évalué 4 scénarios de conception basés sur les stratégies présentées précédemment :

1. Isolation du pilote : un composant `displayAPI` offre une API standard aux développeurs d'applications (e.g. des méthodes `printString` et `clearScreen`). Egalement, un composant `lcd` gere les aspects liés à l'écran et à la communication SPI.
2. Modification de l'API : nous adaptons le composant `lcdAPI` aux besoins de l'application. Quelques mécanismes pour modifier dynamiquement le comportement du pilote sont transformés en propriétés à fixer à la conception.
3. Architecture à couches (i) : le composant `lcd` est décomposé en deux composants gérant les aspects communication SPI et contrôle du LCD séparément.
4. Architecture à couches (ii) : dans la suite de cette démarche, nous encapsulons dans un composant les aspects liés à la façon dont les caractères sont affichés dans l'écran (i.e. la police utilisée).

Comme pour le cas précèdent, nous avons mesuré l'occupation mémoire sous les 4 scénarios proposés. Les résultats sont présentés dans le tableau 8.6.

Analyse des résultats des cas d'étude des pilotes de périphériques. Les résultats obtenus montrent l'impact d'une approche de conception basée sur les composants logiciels sur la performance du logiciel. Cet impact est illustré par les

5. Dans ces deux cas d'étude, les composants n'exhibent aucune interface orientée réflexion.

Scénario		TEXT	DATA	TOTAL	Surcoût
Reference		8060	2530	10590	0,00%
1	Défaut	8534	2640	11174	5,51%
	Optimisé	8136	2546	10682	0,87%
2	Défaut	8160	2640	10800	1,98%
	Optimisé	7828	2546	10374	-2,04%
3	Défaut	8924	2720	11644	9,95%
	Optimisé	8176	2546	10722	1,25%
4	Défaut	9064	2670	11734	10,80%
	Optimisé	8176	2546	10722	1,25%

Table 8.6 – Occupation mémoire (Bytes) du logiciel sous les scénarios proposés - cas écran LCD

différences entre les cas "défaut" et les valeurs de référence dans les tableaux 8.5 et 8.6. En effet, les différents artefacts générés par la chaîne de compilation Fractal/-Think (méta-données, indirections) ont un poids en terme d'occupation mémoire qui peut s'avérer important en fonction des architectures logicielles implémentées.

Les optimisations proposées diminuent fortement cet impact dans tous les scénarios considérés. Lorsque les optimisations agressives sont activées, les indirections sont résolues à la compilation et une grande partie des méta-données ne sont pas générées car elles ne sont plus nécessaires. Les surcoûts assez faibles que l'on observe pour les cas "optimisé" des tableaux 8.5 et 8.6 correspondent à certaines entités qui n'ont pas pu être optimisées.

Par rapport aux stratégies de conception proposées, les optimisations proposées facilitent leur mise en œuvre dans le cadre des systèmes embarqués :

- Nous constatons que le découplage entre l'application et le pilote ne provoque pas un impact fort sur l'occupation mémoire (+2,66% et +5,51% pour le coprocesseur et l'écran LCD, respectivement). Les optimisations proposées réduisent cet impact à des niveaux négligeables.
- Lorsque les décisions de conception impliquent des composants de granularité fine, l'occupation mémoire augmente considérablement. Cela est illustré par les cas "défaut" des scénarios 2 et 3 du pilote du coprocesseur (cf. tableau 8.5), où les granularités fines sont provoquées par l'abstraction des dépendances matérielles du logiciel, avec comme résultat un surcoût de 15,26% et 14,37%, respectivement. Là encore, les optimisations proposées réduisent cet impact à des niveaux négligeables.
- Nous avons constaté au cours de nos évaluations que la composition est responsable d'une bonne partie des surcoûts en mémoire (e.g. du 41,46% du surcoût observé dans le scénario 2 du pilote du coprocesseur). En effet, la composition provoque la création d'interfaces serveur et de liaisons pour assurer la délégation des services. L'optimisation CO1.2, activée par la propriété code[bag=true], rend nul cet impact.
- Nous avons constaté l'impact fort de redefinir les services offerts par les composants en fonction des besoins de l'application. Dans le cas de l'écran LCD (scenario 2) nous avons obtenu une réduction de 2,04% par rapport au scenario de référence, une fois les optimisations appliquées. En général, la mise en place

de cette stratégie donne des résultats satisfaisants au niveau de l'occupation mémoire, au detriment de la réutilisabilité des composants.
- Enfin, nous constatons que les architectures à couches impactent fortement l'occupation mémoire (jusqu'à +20,23%, cf. tableau 8.5). Alors que pour l'écran LCD les optimisations ont diminué considérablement cet impact, pour le pilote du coprocesseur elles n'ont pas été aussi efficaces. Cela s'explique par la "grosseur" des couches logicielles : en effet, dans ce dernier cas les couches sont extrêmement fines, ce qui rend les optimisations inutiles.

En conclusion, les optimisations proposées dans le cadre de cette thèse réconcilient le logiciel de bas niveau avec les approches à base de composants logiciel, afin de tirer profit des avantages espérées et éviter l'utilisation d'approches moins productifs comme celui centré sur le code source.

8.2.4 Cas d'étude : un RTOS à base de composants logiciel

Dans [Loiret 2009] nous avons construit une bibliothèque de composants basés sur le code source de μC/OS-II . Le processus de componentisation mené a tenu compte (i) du dégré de dépendance entre les modules du code source original, afin de séparer au maximum les aspects gestion de tâches et gestion d'évènements, (ii) de la définition des interfaces, (iii) de l'encapsulation des variables globales comme attributs de composants, (iv) de la définition d'attributs de configuration, permettant la construction d'OS sur mesure, et (v) de la définition de composants d'interconnexion (*ressource wrappers*), qui représentent les ressources de l'OS (sémaphores, boites aux lettres, tâches...).

Les retombées d'une telle démarche sur la conception d'applications basées sur μC/OS-II sont :

- Un espace de conception à un haut niveau d'abstraction, comparé aux méthodologies basées sur le code source C. L'utilisation de ressources est faite explicitement au travers des liaisons.
- Flexibilité à la conception : la configuration de l'OS se fait au niveau architectural, via les attributs des composants, et non via des macros `#define` embarquées dans le code source (cf. *legacy* code).
- Flexibilité à l'exécution : l'utilisation du modèle Fractal/Think pour décrire les composants mène à une implémentation qui incorpore les interfaces de réflexivité nécessaires à l'évolution de l'OS à l'exécution. Ce point sera evoqué au chapitre 8.3.1.1.

Les optimisations proposées ici permettent, comme dans le cas des pilotes de périphériques, d'adopter une approche composant sans affecter la performance du logiciel. A ce sujet, nous avons mené deux évaluations concernant l'occupation mémoire (cf. 8.2.4.2) du système et le temps d'exécution d'un chemin d'exécution (cf. 8.2.4.3). Une application représentative du domaine de l'embarqué temps réel a été conçue pour des expérimentations (cf. 8.2.4.1). Aux chapitres suivants nous présentons les résultats obtenus.

8.2.4.1 Application représentative

La figure 8.5 presente l'application qui sert à l'évaluation de l'OS componentifié. Elle est composée de trois tâches ; les tâches `Task1` et `Task2` écrivent et lisent sur une donnée partagée, protégée par un sémaphore binaire. Une tâche qui attend le sémaphore ne peut être bloquée que 4 unités de temps, un paramètre spécifié par le sémaphore. Alors que `Task2` est activée périodiquement par un compteur, `Task1` est activée suite à un évènement externe. Enfin, `Task1` envoie les données lues à la tâche `Task3` qui se charge de les afficher, via une boîte aux lettres.

La figure 8.6 presente l'architecture à composants de l'application, les composants *ressource wrappers* et les dépendances vis-à-vis l'OS, représenté comme un composite. Pour les évaluations qui seront présentées par la suite, nous avons déployé ces composants sur un processeur Pentium 4 à 2 GHz, exécutant un noyau Linux 2.6 et un *patch* temps-réel `Rt-Preempt` [Fu, Luotao and Schwebel, Robert 2010] qui rend le noyau Linux préemptible et permet des mesures temporelles de très haute résolution.

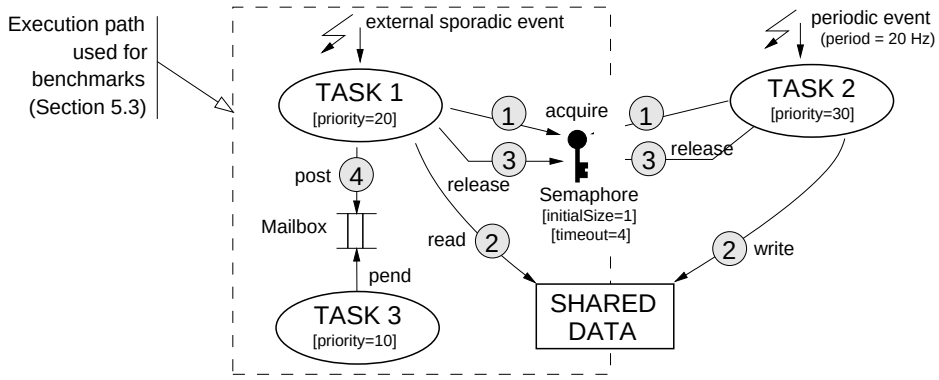


FIGURE 8.5 – Une application representative de l'embarqué temps-réel

8.2.4.2 Occupation mémoire

Le tableau 8.7 présente l'occupation mémoire de la version non-componentifiée de μC/OS-II et du système entier (OS + application), ainsi que le surcoût par rapport à ces valeurs des versions componentifiées : les scénarios *Défaut* et *Optimisé* correspondent aux scénarios 2 et 3 du tableau 8.2, respectivement ; le scénario *Partialement Optimisée* illustre l'optimisation individuelle de composants, où seulement les composants `Task3FunctionalCode` et `SharedData` de la figure 8.6 ne sont pas optimisées comme pour le scénario *Optimisé*.

Nous constatons le surcoût important introduit par l'utilisation d'une approche basée composants (+16,8% pour le RTOS et +20,6% pour le système). Ce surcoût est éliminé par les techniques d'optimisation proposées le long de cette thèse. Le scénario *Partialement Optimisé* illustre les avantages de maîtriser à une granularité

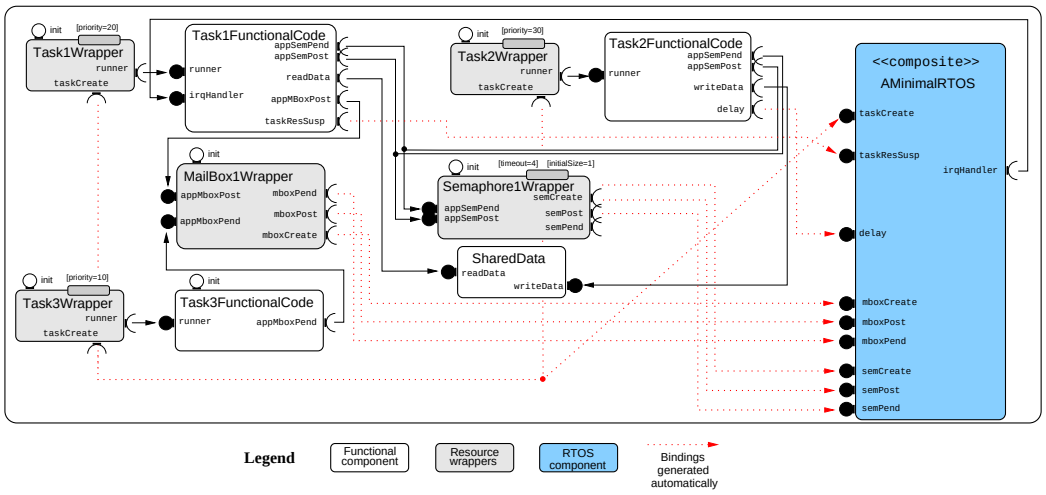


FIGURE 8.6 – Architecture à composants de l'application

assez fine l'optimisation des composants, dans la mesure où le surcoût résultant est négligeable, tout en offrant des possibilités d'évolution.

		Référence	Versions componentifiées		
			Défaut	Optimisé	Part. Optimisée
RTOS	**CODE**	13508	+16,8 %	+0 %	–
	DATA	14072	+3,26 %	+0 %	–
	TOTAL	27580	+9,89 %	+0 %	–
Système entier	**CODE**	14003	+20,6 %	+1,8 %	+2,3 %
	DATA	20252	+4,59 %	+0 %	+0,5 %
	TOTAL	34255	+11,13 %	+0,7 %	+1,2 %

TABLE 8.7 – Occupation mémoire (Bytes) de l'OS et du système complet en sa version non componentifiée, et surcoût des versions componentifiées

8.2.4.3 Temps d'exécution

Le tableau 8.8 présente une comparaison entre la version non-componentifiée et celles componentifiées, par rapport au temps moyen d'exécution du chemin de la figure 8.5. Ce chemin traverse plus de 10 composants du système, tant au niveau applicatif comme au niveau exécutif, puisque les services de sémaphores et des boîtes aux lettres sont concernés. Il considère également un changement de contexte d'exécution entre les tâches `Task1` et `Task2`, implementé par le port de μC/OS-II sur la plate-forme matérielle.

Les résultats obtenus montrent que l'impact sur la performance dans les deux cas componentifiés est faible (+2,8%), même si, en fonction des besoins temporels de l'application, il peut s'avérer non-négligeable. Les optimisations proposées réduisent encore cet impact.

	Référence	**Version componentifiée**	
		Défaut	Optimisé
Chemin d'exécution (μs)	45,97	**+2,8 %**	**+1,3 %**

TABLE 8.8 – Surcoût en terme du temps d'exécution du chemin considéré.

8.2.5 Conclusion sur l'évaluation des optimisations

Les résultats obtenus dans ce chapitre illustrent deux points principaux par rapport aux réifications à l'exécution des composants Fractal/Think :

- Le besoin de mise en place d'une compilation qui prenne compte des exigences au niveau de la performance du logiciel est démontré par les surcoûts non négligeables obtenus avec la chaîne de compilation originale en ce qui concerne la taille de l'empreinte mémoire et le temps d'exécution (cf. cas "défaut" des tableaux 8.3, 8.5, 8.6, 8.7 et 8.8) : de 10,80% à 20,6% en occupation mémoire.
- L'efficacité des techniques d'optimisation proposées au chapitre 6 est démontrée par les surcoûts négligeables obtenus lorsqu'elles sont appliquées. Les résultats dépendent su logiciel examiné. Mais des résultats similaires ont été obtenus pour toutes les couches logicielles définies au chapitre 2.1.2.2 : applicatif, exécutif et pilotes de périphériques.

Dans ce chapitre, les techniques d'optimisation ont été appliqués sur des systèmes qui n'évolueront pas ou peu. C'est un cas courant du domaine des systèmes embarqués, notamment des systèmes fermés au monde extérieur. Pour ce genre de systèmes les techniques d'optimisation proposées permettent l'utilisation d'une approche de conception de haut-niveau, à base de composants logiciel, sans fortement affecter la performance du logiciel.

Pour des systèmes plus ouverts au monde extérieur et aux changements, nous retrouvons le compromis richesse d'évolution vs. performance du logiciel evoqué précédemment dans la thèse. L'application des techniques d'optimisation proposées affectent les capacités d'évolution du logiciel. Au chapitre suivant, nous évaluerons l'infrastructure d'évolution proposée et qui semble adaptée à ce type de systèmes.

8.3 Infrastructures d'évolution

Ce chapitre présente les évaluations menées sur les infrastructures d'évolution proposées pour les systèmes qui correspondent aux catégories d'évolution définies au chapitre 7 de cette thèse. La catégorie "pas d'évolution" n'est pas considérée ici, elle a été l'objet du chapitre précèdent.

Après une étude quantitative de l'impact de l'aspect "évolution" sur la performance du logiciel, dans le cas d'une compilation traditionnelle Fractal/Think pour "Évolution dynamique" et "Évolution statique", au chapitre 8.3.1, nous nous focalisons sur l'évaluation des infrastructures d'évolution proposées pour les systèmes qui correspondent au cas "Évolution non prédéfinie" (cf. 7.2.4 et 7.3), au chapitre 8.3.2 .

8.3.1 Etude de l'impact des Infrastructures d'évolution

Comme cela a été décrit au chapitre 7.1, les infrastructures d'évolution (IdE) par défaut dans Fractal/Think sont constituées de 5 éléments : des méthodes des interfaces d'introspection, des méthodes des interfaces d'intercession, des méta-données propres à ces interfaces, des méta-données sur les entités du modèle à composants et des méthodes d'interfaces de gestion du cycle de vie. La mise en œuvre des IdE proposées pour les catégories "Évolution Dynamique" et "Évolution Statique" (cf. 7.2.1 et 7.2.3, respectivement) implique le déploiement de ces 5 éléments, réifiés de differentes façons.

Notre objectif dans ce chapitre est d'évaluer l'impact du déploiement sur la performance du logiciel. Pour cela, nous évaluons deux cas d'étude : au premier nous ajoutons des capacités d'évolution dynamique au système du chapitre 8.2.4, le deuxième considère un système qui évolue de manière statique.

8.3.1.1 Evolution Dynamique d'un RTOS à base de composants logiciel

Nous avons inclus des interfaces d'introspection et d'intercession, ainsi que les méta-données nécessaires aux IdE qui correspondent à la catégorie Évolution Dynamique, dans les réifications à l'exécution des composants qui constituent l'application représentative du chapitre 8.2.4.1, tant au niveau du RTOS comme au niveau des composants applicatifs.

Le tableau 8.9, issu de [Loiret 2009], présente le surcoût en occupation mémoire par rapport au système de référence. Nous constatons qu'il augmente de 18,6% par rapport aux résultats affichés dans le tableau 8.7. Cela est compréhensible dans la mesure où les deux réifications du système n'offrent pas les mêmes fonctionnalités : l'une peut évoluer à l'exécution alors que l'autre non. Les résultats obtenus donnent pourtant une première idée de l'impact des IdE correspondantes à la catégorie Évolution Dynamique sur l'occupation mémoire des systèmes.

Au chapitre 8.3.2.4 nous déployons des IdE Dynamiques et Non Prédéfinies sur un même système. Dans la mesure où les deux categories d'IdE offrent la même richesse d'évolution, à ce moment là on pourra établir une comparaison entre l'impact de ces deux catégories d'IdE sur la performance su logiciel.

		Référence	**Componentifiée & Évolution Dynamique**
RTOS	**CODE**	13508	+32,2 %
	DATA	14072	+16,8 %
	TOTAL	27580	+24,3 %
Système entier	**CODE**	14003	+47,8
	DATA	20252	+20,9 %
	TOTAL	34255	+31,8 %

TABLE 8.9 – Occupation mémoire (Bytes) de l'OS et du système complet en sa version non componentifiée, et surcoût de la version componentifiée avec des capacités d'évolution dynamique.

8.3.1.2 Evolution Statique d'une application

Afin d'évaluer les IdE qui correspondent au cas d'évolution statique, dans [Navas 2009b] nous avons construit une application d'évaluation basée sur la version componentifiée du cas d'étude presenté au chapitre 8.2.2. Nous avons conçu les scénarios d'évolution décrits dans le tableau 8.10, et généré des IdE adaptées à ceux-ci.

Scénario	Description
0. Pas d'évolution	C'est le scenario de référence, où aucun composant n'évolue et tous les composants sont optimisés agressivement.
1. Evolution Dynamique	Tous les composants exposent les interfaces de réflexivité et les méta-données necessaires, comme au chapitre 8.3.1.1.
2. Evolution Statique – Action d'évolution I	L'action d'évolution I remplace un composant pour un autre du même type mais avec une implémentation différente, et lie le nouveau composant aux composants déjà déployés. L'IdE est générée et optimisée en prenant compte de ces besoins d'évolution du système.
3. Evolution Statique – Actions d'évolution I et II	En plus de l'action d'évolution I, l'IdE est générée et optimisée en prenant compte également de l'action d'évolution II. Celle-ci remplace un composant par un autre qui offre, entre autres, des interfaces serveur du même type que l'ancien. Elle ajoute également un nouveau composant dans l'architecture logicielle, et construit des liens entre les nouveaux composants et ceux déjà déployés.

Table 8.10 – Description des scénarios d'évaluation

Le tableau 8.11 présente l'occupation mémoire du système pour chacun des scénarios. Nous constatons que la connaissance détaillée des activités d'évolution à mener lors de l'exécution du système aux étapes précoces du développement de celui-ci, permet d'optimiser les IdE est de réduire l'impact de l'aspect "évolution" sur l'occupation mémoire. Comparés au cas "Évolution Dynamique", les scenarios 2 et 3 présentent des réductions significatives (-34,50% et -31,43%, respectivement).

Les résultats obtenus pour le scénario 2 illustrent le compromis évoqué précédemment entre la richesse d'évolution et la performance du logiciel. Le surcoût minimal en occupation mémoire (5,24%) est obtenu au detriment de la richesse d'alternatives d'évolution, limitée dans ce cas aux actions décrites au tableau 8.10-2.

La prise en compte des deux actions d'évolution dans la génération de l'IdE (scenario 3), diminue le gain obtenu pour le scénario 2, où seulement une des actions d'évolution a été considérée. L'impact de la mise en place des IdE Statiques sur l'occupation mémoire est lié aux nombre d'actions d'évolution considérées à la conception du système.

Conclusion sur l'impact des IdE Dynamiques et Statiques. Les résultats obtenus pour les deux études de cas de ce chapitre confirment l'impact de l'ajout des capacités d'évolution sur la performance du logiciel, en particulier sur l'occupation mémoire de celui-ci. L'impact grandit lors qu'aucune restriction n'est posée

Scénario	TEXT	DATA	TOTAL
0. Pas d'évolution (référence)	7954	898	8852
1. Evolution Dynamique	12046	2178	14224
surcoût	*51,44%*	*142,53%*	*60,68%*
2. Evolution Statique – I	8294	1022	9316
surcoût	*4,27%*	*13,80%*	*5,24%*
3. Evolution Statique – I et II	8678	1074	9752
surcoût	*9,10%*	*19,59%*	*10,16%*

TABLE 8.11 – Occupation mémoire des scénarios d'évaluation et surcoût des scénarios d'évolution par rapport à la référence (Pas d'évolution).

au niveau des activités d'évolution, sur le déclencheur de l'évolution ni sur le moment auquel le système évoluera, ce qui correspond au cas Évolution Dynamique du chapitre 7.2.1.

La connaissance à la conception des politiques d'évolution, qui correspond au cas Evolution Statique, diminue considérablement cet impact (de 30 à 35%). Néanmoins, le compromis performance du logiciel vs. richesse d'évolution, identifié lors de l'étude de l'état de l'art, reste important.

8.3.2 Infrastructures d'évolution non prédéfinies

Dans ce chapitre nous nous focalisons sur l'évaluation des IdE pour la catégorie des évolutions "Non Prédéfinies". Nous fournissons d'abord, aux chapitres 8.3.2.1 et 8.3.2.2, une description plus détaillée de l'implémentation des agents d'évolution et une estimation de l'occupation mémoire de ces agents sur la plate-forme NXT, respectivement. Nous évaluons ensuite, au chapitre 8.3.2.3, le temps d'exécution des activités d'évolution qui se dérivent de l'invocation des services fournis par les composants miroir. Enfin, nous présentons au chapitre 8.3.2.4 un étude de cas composé d'une application représentative du domaine de l'embarqué et d'un scénario d'évolution assez général, sur lequel nous déployons l'IdE et comparons l'occupation mémoire résultante avec celle obtenu lorsque d'autres catégories d'évolution considérées.

8.3.2.1 Déploiement des agents d'évolution

Au chapitre 7.3.2, nous avons décrit de façon plutôt générale les composants qui matérialisent les liens internes entre les composants miroir et leur contre-partie embarqué, composants que nous appelons *agents d'évolution*. Dans ce chapitre nous nous focalisons sur ces agents embarqués dans le dispositif, i.e. sur les composants `ComDevice` (*in-site*), `EvolutionBackEnd` et `MemoryManager` de la figure 7.6.

L'instantiation de ces composants sur les plates-formes logicielles et matérielles d'évaluation est la suivante :

`ComDevice`. Ce composant est une couche d'abstraction de communication. Il permet d'envoyer et de recevoir des paquets de données avec un contrôle de redondance cyclique (CRC) de 2 bits, sans se soucier du moyen de communication utilisé.

Le `ComDevice` développé peut utiliser indistinctement les périphériques USB et Bluetooth présents dans la plate-forme NXT. `ComDevice` est en fait un composant de type composite, son architecture interne contient des composants de type *ressource wrappers* (cf. 8.2.4) de μC/OS-II , entre autres. Il est composé plus précisément d' :

- un composant μC/OS-II `flag`
- un composant μC/OS-II `sémaphore`
- un composant μC/OS-II `mémoire partagée`
- des composants utilitaires `packet` et `crc`

Le composant `ComDevice`, même s'il est présenté comme faisant partie des composants qui constituent l'IdE, est en réalité ajouté systématiquement au système lorsqu'une application souhaite établir un lien quelconque avec l'extérieur (i.e. dans des systèmes ouverts au changement, cf. 2.1). Pour cette raison il n'est pas considéré dans le calcul de l'occupation mémoire de la réification des liens internes (cf. 8.3.2.2).

`EvolutionBackEnd`. Ce composant est implémenté comme une tâche μC/OS-II qui s'enregistre auprès du composant `ComDevice` pour recevoir les *Messages* provenant du composant `EvolutionFrontEnd`, sa contrepartie *off-site*. Ces *Messages* encapsulent les opérations ponctuelles d'évolution qui doivent être menées sur les réifications embarquées. Par exemple, lorsque la méthode `start` de l'interface serveur `EmbeddedLifeCycleController` du composant `EvolutionFrontEnd` est invoquée (après avoir invoquée la méthode `setComponent` de la même interface, afin de définir le composant que l'on souhaite démarrer), un *Message* spécifique est transmis au composant `EvolutionBackEnd`. Après son réception, `EvolutionBackEnd` lie de façon autonome son interface cliente `LifeCycleController` à celle du composant que l'on désire démarrer, pour ensuite invoquer la méthode `start`.

Nous avons également défini des *Messages* encapsulant des opérations liées à l'écriture et la lecture de la mémoire physique du dispositif, en mode `Byte`, `Word` et `Stream`. Ces *Messages* sont envoyés lors de l'invocation des méthodes des interfaces `AttributeController` et `BindingController` des composants miroir. La correspondance entre les adresses mémoire et les entités du modèle à composants est gérée par les réifications *off-site* (et non par les réifications embarquées) comme décrit au chapitre 7.3.1.3.

`MemoryManager`. Ce composant de type composite gère la mémoire physique du dispositif et offre des services d'écriture et de lecture d'adresses mémoires données, en abstrayant le type de mémoire auxquelles ces adresses font référence. Notre implémentation de ce composant contient deux sous-composants qui permettent d'écrire et lire des deux types de mémoire disponibles dans la plate-forme NXT :

- `RamManager` gère la mémoire volatile RAM. Les opérations d'écriture sont implémentées par des appels à la fonction `memcopy` de la `libC`.
- `FlashManager` gère la mémoire non-volatile FLASH. Cette mémoire est organisée en 1024 pages de taille 256 octets. A cause des caractéristiques physiques de ce type de mémoire, lorsqu'une région dans une page de FLASH doit être réécrite, toute la page doit être réécrite. Le composant `FlashManager` requiert les services du composant `EmbeddedFlashController` (`EFC`), le pilote de la mémoire FLASH.

Tout comme `ComDevice`, le composant `MemoryManager` peut être ajouté au système suite à des besoins fonctionnels ou extra-fonctionnels autres que l'évolution. Contrairement à celui-ci, nous le considérons dans le calcul de l'occupation mémoire de la réification des liens internes, présenté au chapitre suivant.

8.3.2.2 Occupation mémoire des réifications des liens internes

Les agents d'évolution décrits au chapitre précèdent sont des composants des IdE pour la catégorie "Evolution Non Prédéfinie". Il semble pertinent de mesurer l'impact de ces composants sur la performance du logiciel, puisqu'il représente la partie "fixe" de l'impact dû à l'aspect Évolution.

Dans [Navas 2010a] nous avons mesuré l'occupation mémoire due à ces composants [6]. Par rapport aux niveaux d'optimisation appliqués à ces composants, nous avons considéré deux cas : dans le premier cas, les agents d'évolution peuvent eux aussi éventuellement évoluer, et dans l'autre cas ils sont agressivement optimisés. Ces deux cas correspondent aux cas Défaut et Optimisé, 2 et 3 dans le tableau 8.2, respectivement.

Le tableau 8.12 présente les résultats obtenus. Le code source a été compilé avec le niveau d'optimisation `-Os` de `gcc`. La section `RAM TEXT` correspond au code binaire qui doit être déployé en mémoire RAM par des raisons liées aux caractéristiques matérielles de la plate-forme NXT [7].

	Défaut			**Optimisé**		
Composants	DATA	TEXT	RAM TEXT	DATA	TEXT	RAM TEXT
`EFC`	40	0	280	0	0	268
`FlashManager`	280	0	292	256	0	260
`RamManager`	8	20	0	0	12	0
`MemoryManager`	24	352	0	0	292	0
`EvolutionBackEnd`	1090	888	0	1070	832	0
TOTAL	1442	1260	572	1326	1136	528

TABLE 8.12 – Occupation mémoire (en octets) des composants qui matérialisent les liens internes entre les composants miroir et leur contre-partie embarquée.

A partir de ces résultats, on peut constater que :

– La réduction de l'occupation mémoire suite aux optimisations agressives (-8,67% pour l'occupation mémoire totale), bien qu'inférieure à celle obtenue aux chapitres précédents, reste importante et pertinente ; en effet, la plupart des composants ici évalués n'ont pas vocation à évoluer après leur déploiement.

6. Nous avons vu au chapitre précèdent que les composants `ComDevice` et `MemoryManager` peuvent être ajoutés pour satisfaire des besoins non liés à l'évolution. Nous supposons ici que le composant `ComDevice` correspond à ce cas, dans la mesure où les systèmes ouverts aux changements font partie de ceux ciblés par cette thèse (cf. 2.3). En conséquence, il n'est pas considéré dans le calcul de l'occupation mémoire. Par contre, le composant `MemoryManager` y est considéré, puisque nous ne faisons aucune hypothèse sur les besoins du système vis-à-vis la mémoire physique

7. Le code qui est chargé de l'écriture en mémoire FLASH doit être exécuté en récupérant les instructions depuis la mémoire RAM, pour assurer l'intégrité du matériel [Atmel 2010a].

– L'occupation mémoire obtenue (3274 et 2990 octets pour les cas Défaut et Optimisé, respectivement) donne une idée des limitations de cette approche au niveau de l'occupation mémoire des aspects fonctionnels du système. En effet, pour les applications déployées sur des plates-formes extrêmement limitées au niveau de la mémoire physique disponible (e.g. des WSN avec moins de 5 KB disponibles) l'impact sur l'occupation mémoire peut s'avérer prohibitif.
– Néanmoins, l'approche demeure adéquate pour bien d'autres systèmes. En réalité, l'impact de l'inclusion de l'aspect Evolution sur la performance du logiciel devra être évalué au cas par cas, en tenant compte des caractéristiques matérielles et logicielles des systèmes. Au chapitre 8.3.2.4 nous présenterons une évaluation plus approfondie sur une application représentative du domaine de l'embarqué.

8.3.2.3 Exécution des actions d'évolution

Comme presenté au chapitre 7.3, les appels aux interfaces orientées réflexion offertes par les composants miroir se traduisent, à travers des liens internes, en opérations d'évolution menées dans le dispositif embarqué. Dans ce chapitre nous évaluons le temps d'exécution des opérations d'évolution liées à l'invocation de 4 méthodes des composants miroir. Ce temps est mesuré à partir du moment où le composant `EvolutionBackEnd` réceptionne le *Message* qui correspond à l'opération, jusqu'à la fin de son exécution (cf. tableau 8.13).

Itf.	Méthode	Description opération d'évolution résultante
`LCC`	`start`	Invocation, de la part du composant `EvolutionBackEnd`, de la méthode `start` de son interface cliente `LifeCycleController`, qui a été préalablement liée à l'interface serveur du même type du composant que l'on souhaite démarrer.
	`stop`	Invocation, de la part du composant `EvolutionBackEnd`, de la méthode `stop` de son interface cliente `LifeCycleController`, qui a été préalablement liée à l'interface serveur du même type du composant que l'on souhaite arrêter.
`AC`	`setAttValue`	Écriture de valeur en paramètre de la fonction dans l'adresse mémoire où la valeur de l'attribut est stockée à l'exécution, via le composant `MemoryManager`.
`BC`	`bind`	Écriture du descripteur d'une interface serveur dans l'adresse mémoire du descripteur de l'interface cliente du composant, via le composant `MemoryManager`.
`CC`	`addSubComponent`	Écriture d'un nouveau composant (code binaire et données) dans la mémoire physique du dispositif, via le composant `MemoryManager`.

Table 8.13 – Services offerts par les composants miroir et opérations d'évolution résultantes de leur invocation. *Itf.* : interface serveur du composant miroir, LCC : interface `LifeCycleController`, AC : interface `AttributeController`, BC : interface `BindingController`, CC : interface `ContentController`

Afin d'illustrer l'impact des aspects fonctionnels du système sur la performance

liée à l'aspect Évolution, dans nos mesures nous avons considéré deux scénarios : dans le premier le système se trouve en *faible charge*, i.e. la tâche de fond (*idle*) s'exécute environ 90% du temps ; dans le deuxième cas le système a une *charge élevée*, i.e. une tâche fonctionnelle de la plus haute priorité s'exécute environ 90% du temps. Dans les deux cas la tâche du composant `EvoutionBackEnd` est assignée à une priorité moyenne par rapport aux tâches fonctionnelles. L'ordonnanceur de μC/OS-II est appelé tous les 10ms.

Méthodes `start` et `stop` de l'interface `LifeCycleController`. Le temps d'exécution des opérations d'évolution ici considérées dépendent du temps consommé dans l'exécution des méthodes `start` et `stop` du composant que l'on souhaite démarrer ou arrêter. Afin de réduire cette dépendance, nos évaluations ont été effectuées sur des méthodes dont le corp est vide.

Le tableau 8.14 présente les temps d'exécution minimaux, médianes et maximaux obtenus. Compte tenu des caractéristiques de dispersion des séries de données obtenues, nous trouvons que ces statistiques sont les mieux adaptées pour présenter nos évaluations. En particulier, la valeur médiane nous permet d'atténuer l'influence perturbatrice des valeurs extrêmes enregistrées lors de circonstances exceptionnelles issues de l'exécution du logiciel.

	Faible charge		**Charge élevée**	
	`start`	`stop`	`start`	`stop`
Min.	53,66	25,33	53,66	25,33
Médiane	53,83	25,50	53,83	25,50
Max.	171,50	101,33	120,83	117,83

TABLE 8.14 – Temps d'exécution (en μs) des opérations d'évolution : méthodes `start` et `stop` de l'interface `LifeCycleController`

Les résultats obtenus montrent que le temps d'exécution se trouve dans l'ordre des dizaines de μs et que la charge du système n'a pas d'impact sur cette métrique. Dans un cas réel, où le composant que l'on souhaite démarrer ou arrêter doit exécuter un nombre important de commandes afin d'accomplir ces objectif, le temps d'exécution se verra directement affecté.

Méthode `setAttValue` de l'interface `AttributeController`. Comme décrit au chapitre 7.3.1.3, les BinSymbols des attributs contiennent des BinAddresses qui font référence à des types distincts de mémoire physique. C'est le cas pour la plateforme NXT et le `bootloader` par défaut, où un attribut fait référence à deux adresses en mémoire RAM et FLASH.

La modification d'une valeur stockée dans la première affecte immédiatement l'exécution du système, puisque c'est cette adresse à qui l'attribut fait référence à l'exécution. La valeur stockée dans la FLASH fait référence à la valeur de l'attribut au moment du *déploiement* du composant, la modification de cette valeur ne modifie la valeur à l'exécution. Par contre, si le composant passe de l'état "en exécution" à l'état "déployé", lorsqu'il passera à nouveau à l'état "en exécution" le changement

effectué aura un effet sur le comportement du composant. Cela arrive lorsque, par exemple, le dispositif embarqué est mis hors tension pour ensuite être redémarré (valeur persistante).

Nous avons considéré les deux cas d'écriture en mémoire RAM et FLASH. Le tableau 8.15 présente les résultats obtenus.

	Faible charge		Charge élevée	
	RAM	**FLASH**	**RAM**	**FLASH**
Min.	5,33	6055,16	5,33	6110,50
Médiane	5,50	6228,50	5,33	15713,83
Max.	22,33	6920,00	14,33	26060,16

TABLE 8.15 – Temps d'exécution (en μs) des opérations d'évolution : méthode `setAttValue` de l'interface `AttributeController`

Nous remarquons que le temps d'exécution de l'écriture d'une valeur dans une adresse physique dépend fortement du type de mémoire physique à laquelle l'adresse fait référence. Le temps d'écriture en FLASH est ostensiblement supérieur à celui en RAM. Cela s'explique par le fait de devoir écrire toute une page de mémoire (256 octets) lorsque le changement concerne une région de celle-ci, dans ce cas 4 octets (la taille d'une valeur de type `int`).

Nous constatons à nouveau que le changement de charge n'a aucun impact sur le temps d'exécution lorsque celui-ci se trouve dans l'ordre de μs, comme c'est le cas de l'écriture en RAM. Par contre, pour l'écriture en FLASH, la valeur médiane du temps d'exécution à charge élevée est le double que pour le cas faible charge (15713,87 vs. 6228,50 μs).

Afin de mieux identifier la cause d'une telle différence, la figure 8.7 présente l'exécution des tâches *idle* et `EvoutionBackEnd`, ainsi que de la tâche utilisateur à qui est assignée la plus haute priorité à charge élevée. Le temps consommé lors de 3 modifications consécutives de valeurs d'attributs est représenté par des lignes horizontales.

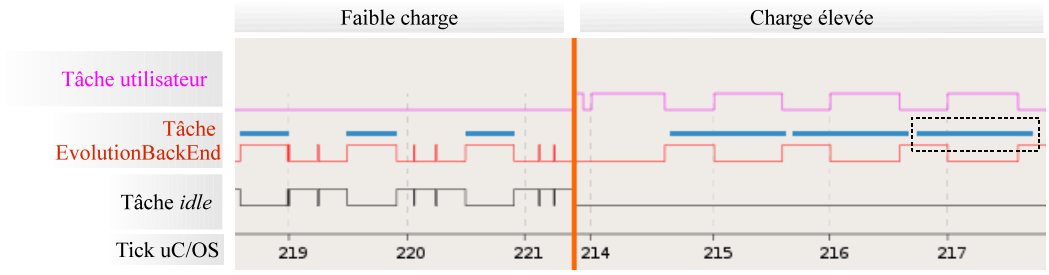


FIGURE 8.7 – Exécution des tâches lors de trois modifications consécutives de valeurs d'un attribut (cas écriture en FLASH), à charges faible et élevée.

Nous remarquons que la modification de la valeur de l'attribut, menée par la tâche `EvoutionBackEnd`, est interrompue par la tâche utilisateur de haute priorité.

Cela vient du fait que lors de l'écriture d'une page de FLASH les interruptions sont bloquées pour ensuite être débloquées juste après l'écriture de la page, afin de ne pas perturber de trop le système. Ainsi, quand les interruptions sont réactivées, un tick système peut se produire et éventuellement l'ordonnanceur est appelé. Comme il y a une tâche avec une plus haute priorité qui demande s'exécuter, la tâche `EvolutionBackEnd` est préemptée sans avoir pu finir d'autres opérations requises par l'action d'évolution, comme la verification d'absence d'erreurs d'écriture. Celles-ci devront être finies quand l'ordonnanceur redonne la main à la tâche `EvoutionBackEnd`.

Méthode `bind` de l'interface `BindingController`. La mise en place d'une liaison dirigée d'une interface cliente vers une interface serveur, se fait via l'écriture d'une valeur qui correspond au descripteur de l'interface serveur, dans l'adresse mémoire où le descripteur de l'interface cliente est stocké.

Par conséquent, le temps d'exécution de cette action est le même que celui lié à la modification de la valeur d'un attribut via la méthode `setAttValue` (cf. tableau 8.15). Tout comme pour le cas des attributs, la modification de l'adresse en mémoire FLASH correspond au cas où la liaison est persistante.

Méthode `addSubComponent` de l'interface `ContentController`. A la différence des précédentes méthodes, l'ajout d'un composant se fait par l'envoie de n *Messages* d'écriture en mode *stream* , chacun contenant un fragment de la réification binaire du composant. Le code et les données des $n-1$ premiers *Messages* sont sauvegardés temporairement dans une mémoire tampon dédiée. Seulement après la réception du n-ième paquet l'écriture du composant en RAM ou en FLASH démarre.

Nous avons evalué l'ajout d'un composant de taille totale 400 octets à travers de $n = 7$ *Messages*. Le tableau 8.16 présente le temps consommé dans la réception du n-ième *Message* et l'écriture du composant dans la mémoire physique. Le temps consommé par la réception et stockage des $n-1$ premiers paquets est de 15,3 μs en moyenne sous les deux niveaux de charge.

	Faible charge		**Charge élevée**	
	RAM	**FLASH**	**RAM**	**FLASH**
Min.	31,50	12121,83	31,50	31469,33
Médiane	36,00	12316,83	31,66	31477,16
Max.	49,00	12585,00	49,00	31938,16

TABLE 8.16 – Temps d'exécution (en μs) des opérations d'évolution : méthode `addSubComponent` de l'interface `ContentController` – réception du dernier *Message* est écriture en mémoire physique.

Concernant l'ajout du composant en mémoire FLASH, nous observons un phénomène similaire à celui observé dans le cas de la modification de la valeur d'un attribut, plus marqué ici à cause de la quantité de données à écrire. En effet, l'écriture de 400 octets en mémoire FLASH implique la réécriture d'au moins deux pages (2 x 256 octets) de mémoire. A charge élevée, l'action d'évolution est

interrompue par les tâches plus prioritaires, ce qui provoque un temps d'exécution supérieur.

Conclusion sur l'évaluation des temps consommé par les opérations d'évolution. Nous remarquons que l'exécution d'actions d'évolution qui impliquent l'écriture de données en RAM perturbe assez peu le fonctionnement du système : le temps d'exécution est de l'ordre de quelques μs, sans grande variation par rapport à la charge. D'autre part, la modification des valeurs de la mémoire RAM affecte de manière immédiate le comportement du système. Des mesures doivent être prises afin d'assurer l'intégrité du système pendant et après l'opération d'évolution. Ce point sera évoqué au chapitre suivant.

L'exécution des opérations d'évolution qui impliquent l'écriture de données en FLASH, quant à elle, perturbe assez fortement le fonctionnement du système. Cela vient du fait que (i) les appels à l'ordonnanceur sont retardés lors de l'écriture en mémoire FLASH, puisque les interruptions matérielles doivent être suspendues à ce moment à cause des contraintes de la plate-forme matérielle, et (ii) la modification d'une zone mémoire de la FLASH peut entrainer la réécriture de plusieurs pages mémoire, dans la mesure où la granularité d'écriture de ce type de mémoire est d'une page (256 octets, cf. 2.1.2.1). Cette perturbation issue des actions d'évolution peut entrainer des comportements hasardeux.

Enfin, nous constatons l'influence de la cadence de l'ordonnanceur sur le fonctionnement du système et en particulier sur l'exécution des opérations d'évolution. Les résultats ici présentés ont été obtenus avec une période de 10ms. Lorsque nous avons fait des évaluations avec une cadence de 100ms nous avons constaté que les tâches dépendantes de données entrantes (avec des queues, par exemple) comme celle de l'`EvolutionBackEnd`, ont généralement plusieurs entrées à traiter. Ainsi les tâches, lorsqu'elles sont à l'état actif peuvent traiter plus de données, augmentant la stabilité du système. Néanmoins, le temps de réaction des tâches critiques peut être affecté par cette augmentation de la cadence de l'ordonnanceur.

8.3.2.4 Etude de cas

Nous avons conçu une application basée sur des composants Fractal/Think, qui contient une bonne partie des composants inclus dans la bibliothèque des composants développée pour la plate-forme NXT : HAL, pilotes de périphériques et exécutif temps réel μC/OS-II (cf. 8.1.3). Les objectifs que nous poursuivons en développant cette application réelle sont :

- Évaluer de façon qualitative la richesse d'évolution des IdE "Non Prédéfinies", notamment la possibilité de faire évoluer le comportement du système après son déploiement initial, en exécutant des actions d'évolution non prévues à la phase de conception du système.
- Étudier la façon dont les composants miroir et les composants qui matérialisent les liens internes, s'intègrent au système, lors des phases de *conception*, *mise en œuvre* et *exécution*.
- Estimer l'impact sur la performance de cette application réelle, concrètement sur son occupation mémoire, de l'inclusion d'une IdE du type "Non Prédé-

finie". Comme décrit au chapitre 8.3.2.2, l'impact de l'inclusion de l'aspect Évolution sur la performance du logiciel doit tenir compte des caractéristiques matérielles et logicielles des systèmes.
– Comparer cet impact avec celui obtenu lorsque des IdE Dynamiques et Statiques sont ajoutées à ce système.

Description du système. L'application développée exécute de manière concurrente 3 tâches représentées par 3 composants : `Task1`, `Task2` et `Task3`. `Task1` réagit à un évènement externe et sporadique, l'appui de l'un des boutons du NXT, qui détermine l'augmentation ou diminution d'une valeur qui est envoyée au composant `Task2` via une boite aux lettres. `Task2` à son tour utilise les services du pilote de son de la plate-forme pour émettre de façon périodique un son de fréquence égale à la valeur reçue de `Task1`. `Task3` est synchronisée avec `Task1` à travers d'un sémaphore, elle envoie périodiquement des données à l'extérieur du dispositif via le composant `comDevice` et le périphérique USB.

Les pilotes de périphériques sont pour la plupart regroupés dans un composite `BSP` (*Board Support Package*). Trois tâches supplémentaires sont exécutées à l'intérieur de ce composite : elles gèrent les boutons et la mise hors tension, l'écran LCD et la communication USB, respectivement. Également, les composants μC/OS-II sont regroupés dans un composite `uCOS`.

La figure 8.8 presente une vision architecturale d'un sous-ensemble des composants du système (en noir). Le tableau 8.17 présente à son tour une synthèse des caractéristiques du système en sa totalité. Le nombre de composants qui s'y exécutent, les fonctionnalités décrites et la complexité résultante, font que le système nous apparait représentatif du domaine de l'embarqué visé par notre étude.

Entités du modèle	
Composants	45
Primitifs	35
Composites	10
Attributes	44
Interfaces	417
Serveur	190
Clientes	227
Liaisons	224
Abstractions μC/OS-II	
Tâches	6
Sémaphores	3
Mémoires partagées	2
Queues	2
Boites aux lettres	1

TABLE 8.17 – Caractéristiques du système – nombre d'entités du modèle à composants et d'abstractions μC/OS-II

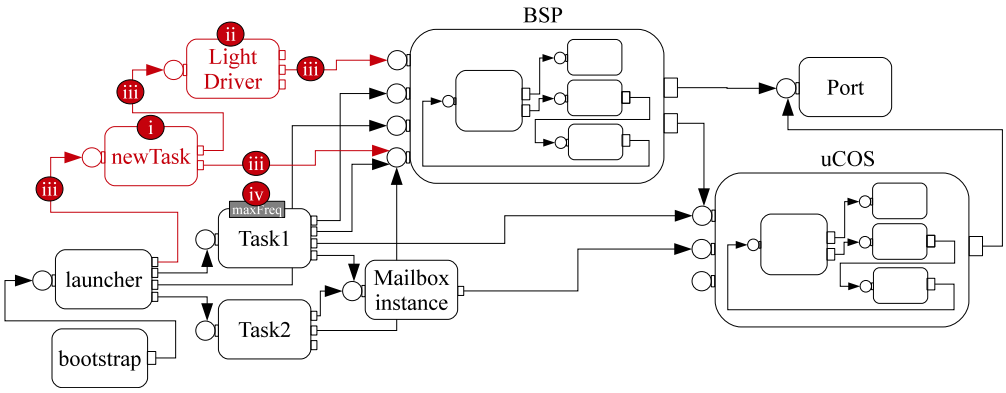


FIGURE 8.8 – Extrait de la vue architecturale du système original de l'étude de cas du chapitre 8.3.2.4 (composants en noir). En rouge les nouveaux composants et liaisons suite aux actions d'évolution.

Actions d'évolution considérées. Nous décrivons maintenant les actions d'évolution menées sur ce système. Nous avons conçu un scénario d'évolution suffisamment général composé de 4 actions exécutées séquentiellement, dont le résultat est illustré par des chiffres notés entre parenthèses ((i) à (iv)) dans la figure 8.8 :

i. L'ajout du composant `newTask`, qui est en fait un composite qui contient deux composants primitifs représentant le code exécutable de la nouvelle tâche et le *ressource wrapper* d'une tâche μC/OS-II .

ii. L'ajout du composant `LightDriver`, qui est le pilote d'un dispositif externe qui peut être connecté à la plate-forme NXT via l'un des ports utilisés pour connecter les capteurs. Ce dispositif est un indicateur basé sur deux leds.

iii. La résolution des dépendances de ces deux nouveaux composants vis-à-vis les composants précédemment déployés :
   - Le composant `LightDriver` requiert du composant `BSP` l'accès à l'un des ports entrée/sortie de la plate-forme matérielle.
   - Le composant `newTask` requiert les services d'affichage du composant `LightDriver`, ainsi que d'autres services des composants `BSP` et μ`COS` comme la lecture des boutons et la mise en attente (*delay*).
   - Afin de démarrer la tâche du composant `newTask`, le composant `launcher` doit être lié à celui-ci.

   Ces dépendances sont résolues en créant dynamiquement des liaisons entre les composants.

iv. La modification de la valeur de l'attribut `maxFreq` du composant `Task1`, qui détermine la fréquence maximale du son périodique émit par le dispositif.

Chacune des actions d'évolution ici considérées est menée par l'exécution d'une série d'activités d'évolution parmi celles décrites au chapitre 8.3.2.3, qui à son tour déclenchent une ou plusieurs opérations d'évolution dans le dispositif embarqué. Par exemple, pour l'action (iv), la modification de la valeur d'un attribut, déclenche les activités suivantes :

- la création d'une liaison entre le composant `EvolutionBackEnd` et le composant `Task1` via leur interface de type `LifeCycleController` cliente et serveur, respectivement ;
- l'invocation de la méthode `stop` du composant `Task1`, afin d'arrêter son exécution pour modifier de manière sûre la valeur de son attribut ;
- la modification de la valeur de l'attribut `maxFreq`, pour sa réification à l'exécution (valeur stockée en mémoire RAM) et pour sa réification au déploiement (valeur stockée en mémoire FLASH) ;
- l'invocation de la méthode `start` du composant `Task1`, afin de redémarrer son exécution.

Notons que nous demandons l'arrêt total du composant `Task1` avant la modification de la valeur de l'attribut `maxFreq`. De même, pour certaines actions d'évolution nous demandons l'arrêt total du système. Cette mesure radicale est demandé afin d'assurer l'intégrité de l'exécution globale du système. Néanmoins, dans la mesure où les composants offrent des services qui permettent la réduction du *down-time* du système, par exemple dans l'implémentation des méthodes des interfaces `ReconfigurationController` (cf. tableau 7.1), l'infrastructure d'évolution proposée sera en mesure d'invoquer les méthodes adéquates [8].

Résultats obtenus et analyse. Le compilateur Fractal/Think génère des réifications miroir pour chacun des 45 composants du système. Ces réifications offrent des implémentations par défaut des interfaces serveur décrites au chapitre 7.3.1.1 : `LifeCycleController`, `ReconfigurationController`, `BindingController`, `AttributeController`, `ContentController`, `ComponentIdentity` et `ComponentContent`. A l'exécution, ces composants miroir sont liés aux réifications des composants qui font référence à des étapes antérieures du cycle de vie (également générées par la chaîne de développement), ainsi qu'aux composants qui matérialisent les liens internes (du coté *off-site*) et qui avaient été construits précédemment : `EvolutionFrontEnd` et `ComDevice`.

L'implémentation des composants embarqués `EvolutionBackEnd` et `MemoryManager` dépend fortement de la plate-forme logicielle et materielle sous-jacente, ils ont été ajoutés à la bibliothèque de composants construite pour la plate-forme NXT. A la conception du système, ces composants s'intègrent assez facilement aux composants fonctionnels, puisque le couplage entre les deux groupes des composants est minime et bien défini : les composants de le tableau 8.12 requièrent quelques services basiques de l'exécutif (*delay*, sections critiques...) et un moyen de communication avec l'exterieur, déjà inclut dans l'application via le composant `ComDevice` (*in-site*). La tâche qui gère l'évolution du système

8. En termes généraux, nous ne faisons aucune hypothèse par rapport à la gestion de l'état de la réification embarquée des composants, mis à part le fait de supposer que l'invocation des méthodes `start` et `stop` de l'interface `LifeCycleController` démarre et arrête complètement l'exécution du composant. En particulier, nous ne faisons aucune hypothèse sur les algorithmes implémentés par le composant afin d'atteindre un état où l'évolution soit possible (état *quiescent*), par exemple le comptage de *threads* [Helander 1998] ou les intercepteurs dynamiques [Soules 2003] ; dans [Polakovic 2006] ces deux algorithmes sont utilisés dans la définition des méthodes de l'interface `ReconfigurationController`.

est démarrée en liant les composants `launcher` et `EvolutionBackEnd` via leurs interfaces du type `LifeCycleController`, cliente et serveur, respectivement.

Afin de comparer l'impact de l'ajout des IdE étudiées tout au long de cette thèse sur l'occupation mémoire du système, nous avons évalué 4 scénarios qui correspondent aux 4 catégories d'évolution définies au chapitre 7 :

1. Pas d'évolution : *tous* les composants sont embarqués et *aucun* des composant du système n'offre des fonctionnalités d'évolution. Ils sont tous optimisés agressivement (cas 3 dans le tableau 8.2). Ce scénario est celui de référence dans cette évaluation.
2. Évolution Dynamique : *tous* les composants du système offrent les interfaces de réflexion et tous les méta-données nécessaires sont également générées et ajoutés aux réifications embarquées, i.e. l'IdE dynamique est générée pour toutes les entités du modèle dans le système.
3. Évolution Statique + partialement optimisé : des IdE statiques sont générées seulement pour les composants qui sont concernés par les actions d'évolution définies précédemment. Les composants qui ne sont pas concernés sont optimisés agressivement (cas 3 dans le tableau 8.2).
4. Évolution Non Prédéfinie : des IdE non définies sont générées pour *tous* les composants du système, c'est le cas traité tout au long de ce chapitre 8.3.2.

Le tableau 8.18 présente les résultats obtenus. Nous constatons à nouveau l'impact des IdE dynamiques sur l'occupation mémoire (+141%), encore plus marqué ici par le nombre important de composants. La mise en place des IdE statiques et l'optimisation des composants non concernés par les actions d'évolution réduisent considérablement cet impact, mais la richesse d'évolution est limitée dans la mesure où seulement les actions d'évolution définies à la conception peuvent être exécutées.

Categorie IdE	**TEXT**	**DATA**	**TOTAL**
Pas d'évolution	31648	5893	37541
Evolution Dynamique	63960	26713	90673
surcoût	*353,30%*	*65,72%*	*141,53%*
Evolution Statique + part. optimisé	41056	9157	50213
surcoût	*29,72%*	*55,38%*	*33,75%*
Evolution Non Prédéfinie	37008	7509	44517
surcoût	*16,93%*	*27,42%*	*18,58%*

Table 8.18 – Occupation mémoire du système dans les scénarios définis.

L'impact des IdE non prédéfinies est très inférieur à celui des IdE dynamiques (-50,90% par rapport au dernier). Ce résultat est obtenu sans sacrifier les capacités d'évolution du système, puisque tous les entités du modèle à composants peuvent évoluer à l'exécution grâce aux composants miroir et aux liens internes.

Nous pouvons estimer l'impact de l'inclusion de l'aspect Évolution sur le système pour les scénarios 2, 3 et 4, en calculant la différence entre les résultats obtenus et ceux du scénario 1. Pour les IdE non prédéfinies, cette différence est de 6976 octets au total. Ce chiffre résulte aussi de l'addition de l'occupation mémoire due

aux agents d'évolution (2990 octets, cf. tableau 8.12) et des méta-données générées pour les entités du modèle, soit 3986 octets. Le ratio a donc tendance à diminuer avec la taille du système. Mais il augmente évidemment lorsque la granularité des composants inclus dans le système est fine. Au cours des mesures, on voit un lien fort entre nombre de composants et surcoût mémoire.

8.3.3 Conclusion sur l'évaluation des IdE

Les résultats obtenus dans ce chapitre soulèvent les constatations suivantes :

- La solution par défaut de Fractal/Think par rapport au besoin d'évolution du logiciel embarqué est celle d'assigner la catégorie d'Évolution Dynamique à toutes les entités du modèle à composants et de les réifier en fonction. Cette solution provoque un surcoût considérable, en termes relatifs et absolus, sur la performance du logiciel, plus particulièrement sur l'occupation mémoire du système.
- La mise en place des infrastructures d'évolution proposées pour les entités appartenant à la catégorie "Évolution Statique", réduit ce surcoût. En effet, l'application ponctuelle des techniques d'optimisation proposées au chapitre 6 en fonction de la connaissance précoce des propriétés des actions d'évolution qui seront menées à l'exécution du système (*qui*, *comment* et *quand*) permettent de minimiser cet impact. Néanmoins, le compromis qui est soulevé entre la performance du logiciel et la richesse d'évolution, limite cette solution. Ce phénomène avait déjà été observé lors de l'étude de l'état de l'art de cette thèse (cf. 3.3).
- L'assignation de la catégorie "Évolution Non Prédéfinie" aux entités du modèle à composants permet de conserver la richesse d'évolution du système avec un impact sur l'occupation mémoire inférieur. De plus, l'intégration des agents d'évolution et des composants miroir avec le système au moment de sa conception et à son exécution reste simple à mettre en œuvre.
- Concernant l'exécution des opérations d'évolution qui composent les activités d'évolution du système, la plupart de ces opérations impliquent des actions ponctuelles et/ou la modification de zones mémoire de petite taille. Le temps consommé par ces actions est de l'ordre de quelques μs, il ne perturbe pas le fonctionnement du système, même lorsqu'il se trouve chargé.
- L'ajout de nouveaux composants dans le système déjà déployé est plus couteux. Le temps consommé par cette action est important et peut perturber l'exécution du système. Dans la mesure où des techniques pour minimiser le *down-time* et pour assurer l'intégrité du système ont été prévues à la conception de celui-ci, ces actions peuvent être menées.
- L'occupation mémoire due aux agents d'évolution peut éventuellement rendre l'approche prohibitif sur des systèmes extrêmement limités en mémoire physique disponible. Outre les techniques d'optimisation à la compilation proposées, nous n'avons pas optimisé les implémentations des agents d'évolution, ce qui donne un certain marge de manœuvre à ce sujet. De plus, les opérations d'évolution disponibles pourraient être limitées afin de réduire l'occupation mémoire due aux agents d'évolution.

Chapitre 9

Conclusion et Perspectives

Sommaire

9.1 Bilan des contributions 175
9.2 Perspectives 177

Ce chapitre clôture la thèse. Nous commençons par rappeler la problématique considérée et nous présentons une synthèse de nos contributions. Ensuite, nous proposons un certain nombre de perspectives à plus ou moins long terme de nos travaux.

9.1 Bilan des contributions

Rappel de la problématique. Cette thèse s'intéresse à la partie logicielle d'un sous-ensemble de systèmes embarqués qui possèdent un haut degré d'interaction avec des agents externes au système et sont très ouverts aux changements de leur comportement lors de son exécution. De plus, leur développement doit prendre en compte de fortes contraintes au niveau de l'utilisation de ressources physiques. Le modèle économique actuel du développement de logiciel embarqué pour ce type de systèmes impose un certain nombre de besoins parmi lesquels nous soulignons :

- l'existence d'un modèle clair et fidèle de la structure du logiciel embarqué qui facilite la gestion de la complexité du logiciel ;
- la capacité du logiciel embarqué à évoluer à l'exécution, i.e. d'adapter son comportement aux changements de son environnement.

Nous remarquons que les méthodologies de développement basées sur les *composants logiciels* répondent de manière générale à ces exigences, et elles sont par conséquent à la base des contributions de cette thèse. Néanmoins, nous constatons que les solutions existantes n'arrivent pas à concilier ces exigences avec la prise en compte de contraintes physiques et notamment des contraintes sur l'utilisation des ressources matérielles limitées (la mémoire, les processeurs et les médiums de communication).

D'un coté, les approches qui utilisent les composants pour faciliter l'implémentation et l'exécution des activités d'évolution, impactent fortement la performance du logiciel embarqué, notamment sur son occupation mémoire et son temps d'exécution. D'un autre coté, les approches basées sur des composants logiciel orientées au domaine de l'embarqué appliquent, de manière systématique, des optimisations agressives sur la « glue », i.e. le code généré à la compilation qui assure l'interopérabilité des composants. Mais ce choix limite, et dans certains cas interdit, l'évolution du système après son déploiement.

Synthèse des contributions. Dans cette thèse nous nous appuyons sur une plate-forme existante de développement basé sur des composants logiciels, Fractal/-Think. ce choix est lié aux capacités d'administration et d'évolution offertes par le modèle Fractal et à l'applicabilité de Fractal à l'embarqué au travers de son implémentation pour le langage C, Think.

Notre démarche peut se résumer en deux étapes. D'abord nous nous intéressons à l'optimisation de la « glue » générée par la chaîne de compilation Fractal/Think ; ce choix est motivé par l'intention de rester relativement indépendants des caractéristiques des applications, très hétérogènes dans le domaine de l'embarqué. Ensuite, cette maîtrise de la génération de code est utilisée pour générer des infrastructures d'évolution optimisées et adaptées aux besoins du logiciel en termes de ses capacités d'évolution. Les contributions de cette thèse sont :

- Nous proposons des techniques d'optimisation qui peuvent être appliquées de manière indépendante sur chacune des instances des entités du modèle à composants : attributs, interfaces serveur, interfaces clientes, liaisons, composants composites et primitifs. Par ailleurs, nous définissons des niveaux d'optimisation pour chacune des entités, qui couvrent tout le spectre des optimisations, en termes de leur agressivité (génération ou pas des services d'évolution, génération ou pas des méta-données). Ainsi, nous sommes en mesure de générer, à partir des descriptions de haut-niveau des composants, plusieurs versions de leurs représentations à l'exécution qui mènent à des profils variés de performance du logiciel, notamment en ce qui concerne l'occupation mémoire et le temps d'exécution.

  Cet enrichissement des options d'optimisation des composants permet de régler plus finement la performance du système en fonction de l'ensemble des besoins fonctionnels et extra-fonctionnels, parmi lesquels on trouve les capacités d'évolution.

- Nous définissons quatre catégories d'évolution qui peuvent être assignées à chaque instance des entités du modèle à composants. L'une de ces catégories correspond au cas trivial où l'on sait à la conception que l'entité n'évoluera pas après son déploiement. Les autres catégories se différencient entre elles en fonction des réponses à des questions simples sur le "comment" et le "quand" de l'évolution : quelles sont les politiques d'évolution de l'entité ? à quel instant précis de l'exécution l'entité évoluera ? qui déclenche les actions d'évolution ? à quel moment du cycle de vie de l'entité ces informations sont disponibles ?

  Ainsi, la catégorie "Pas d'évolution" est assignée aux entités non plausibles d'évoluer à l'exécution ; la catégorie "Évolution dynamique" aux entités pour qui on sait à la conception qu'elles évolueront à l'exécution, mais on connait pas la façon dont elles le feront ; la catégorie "Évolution statique" aux entités pour qui on sait à la conception qu'elles évolueront à l'exécution, et pour qui on connait la façon dont elles évolueront ; et la catégorie "Évolution non prédéfinie" aux entités pour qui on sait à la conception qu'elles *peuvent* évoluer, mais ce n'est qu'après le déploiement qu'on sait si elles *vont* évoluer.

  En fonction des catégories assignées, nous mettons en place des infrastructures d'évolution adaptées aux besoins de chaque entité en termes de sa richesse

d'évolution, et couplées avec les besoins d'optimisation du logiciel.

- Nous introduisons la notion de *réification* de composant, une collection de données et de comportements qui encapsulent un ou plusieurs aspects spécifiques du composant à un moment précis de son processus de développement. Dans notre approche, les réifications assurent une continuité dans le cycle de vie de développement des composants, indispensable dans la mesure où nous agissons à deux moments du cycle de vie (compilation et exécution) sur deux aspects extra-fonctionnels entremêlés (optimisation et évolution, respectivement). Dans l'architecture proposée, chaque composant possède au moins une réification qui est créée à chaque étape de son cycle de vie. Les activités du cycle de vie du logiciel sont focalisées sur la construction de ces réifications, et pour cela on établit des liens entre réifications du même composant à différents moments de son cycle de vie.

  La mise en œuvre des réifications nous permet de maîtriser le compromis entre la richesse d'évolution et la performance du logiciel.

La pertinence et la validité de nos contributions a été illustrée à travers des évaluations menées au chapitre 8 de cette thèse.

Concernant l'optimisation, l'efficacité des techniques proposées est démontrée par les surcoûts négligeables obtenus lorsqu'elles sont appliquées. Nous avons mis à l'épreuve ces techniques sur différentes couches qui composent le logiciel embarqué : les applications, les exécutifs et les pilotes de périphériques, pour des plates-formes matérielles différentes.

Concernant les infrastructures d'évolution, nous avons constaté la manière dont ces infrastructures sont générées en fonction des besoins en termes d'évolution des entités, minimisant l'impact sur la performance du logiciel. Nous avons également mis en œuvre l'architecture de développement basées sur les réifications des composants, confirmant son efficacité et la viabilité de l'approche.

9.2 Perspectives

Les résultats obtenus lors de ce travail de thèse permettent d'entrevoir un certain nombre de perspectives de recherche :

- Une première perspective possible est liée à une meilleure intégration du concept de *réification* dans le modèle à composants Fractal/Think. Pour l'instant, cette intégration se fait dans la chaîne de compilation Fractal/-Think, plus concrètement lors de la réécriture et la génération de code (cf. chapitre 2.2.4.2 page 29), i.e. à une étape relativement tardive du cycle de vie de développement des composants. Il serait intéressant de tenir compte de la notion de réification lors de la conception des composants primitifs et des assemblages de composants, ce qui servirait à la mise en œuvre des activités d'évolution (en général, pas forcement l'évolution à l'exécution) du logiciel.

  Idéalement, via une interface graphique appropriée il serait possible de tracer la façon dont une modification sur la représentation de plus haut-niveau d'un composant impacte les différentes réifications de ce composant tout au

long du cycle de vie, et la façon dont les différentes aspects fonctionnels et extra-fonctionnels (parmi ceux-la, l'optimisation et l'évolution) participent à la construction de ces réifications. Il pourrait être evisagé d'arriver jusqu'au point où une modification sur la documentation technique d'un composant provoque des modifications sur le composant en exécution dans un dispositif embarqué.

- Une deuxième perspective concerne les limitations que l'on rencontre dans les chaînes de compilation de composants pour l'embarqué, et en particulier dans Fractal/Think. Nous avons constaté au chapitre 7.2.3 que certaines des optimisations que l'on pourrait appliquer sur les entités auxquelles on assigne la catégorie Évolution Statique reposent sur l'analyse du comportement des composants et de celui des infrastructures d'évolution, et moins sur les aspects structuraux des assemblages de composants. Pourtant, la chaîne de compilation Fractal/Think n'effectue aucun type d'analyse statique sur le code source C fourni par les développeurs, et donc elle n'est pas capable de prendre en compte le comportement des composants. La mise en place d'une telle analyse intégrée à la chaîne de compilation où la mise en œuvre de notre approche dans une plate-forme qui offre des telles fonctionnalités, comme TinyOS (cf. chapitre 2.2.4.2 page 23), sont donc des pistes de recherche à envisager.
- En lien avec la perspective précédente, une troisième perspective concerne une meilleure intégration avec les compilateurs de bas-niveau. Les optimisations proposées, comme celles des plates-formes de référence, sont implémentées lors d'une première phase de compilation (cf. figure 3.1 page 42), la génération de code binaire est réalisée par les compilateurs de bas-niveau tels que GCC. Une meilleure maîtrise de cette dernière phase dans la construction des réifications à l'exécution permettrai de régler plus finement le compromis entre la richesse d'évolution et l'optimisation du logiciel, et d'offrir des optimisations sur les assemblages de composants encore plus efficaces.
- Enfin, une quatrième perspective est en lien avec le cadre défini pour cette étude, en ce qui concerne le logiciel embarqué. Par des raisons liées à l'état actuel de l'industrie du développement de logiciel embarqué et aux fortes limitations au niveau des ressources physiques disponibles, nous nous sommes focalisé sur les systèmes embarqués dont le code est écrit en langage C, compilé en langage assembleur et ensuite exécuté sur un boîtier de type microcontrôleur. Rien ne nous empêche pourtant d'étendre notre approche basée sur les réifications des composants à d'autres plates-formes de développement où l'exécution est assurée par des machines virtuelles et où le codage des composants est réalisé à un plus haut-niveau d'abstraction. Par ailleurs, de plus en plus de plates-formes de développement basées sur des langages tels que Java sont proposées, tant dans le contexte académique qu'industriel, ce qui marque une tendance qu'il serait pertinent d'étudier.

Annexe A

Publications liées à la thèse

Pendant les travaux de thèse, l'auteur a participé à la rédaction des publications suivantes, présentées dans des confèrences internationales :

- Juan F. Navas, Jean-Philippe Babau et Jacques Pulou. *An Optimized Run-time Evolution Infrastructure for Component-Based Embedded Systems.* In Proceedings of the ninth BElgian-NEtherlands software eVOLution seminar, BENEVOL '10, 2010.
- Juan F. Navas, Jean-Philippe Babau et Jacques Pulou. *A component-based run-time evolution infrastructure for resource-constrained embedded systems.* In Proceedings of the ninth international conference on Generative programming and component engineering, GPCE '10, pages 73–82, New York, NY, USA, 2010. ACM.
- Juan F. Navas, Jean-Philippe Babau et Olivier Lobry. *Minimal yet effective reconfiguration infrastructures in component-based embedded systems.* In Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime (SINTER'09), pages 41–48, New York, NY, USA, 2009. ACM.
- Juan Navas et Jean-Philippe Babau. *Efficient and Adapted Component-Based Strategies for Embedded Software Device Drivers Development.* In Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 02, CSE '09, pages 514–519, Washington, DC, USA, 2009. IEEE Computer Society.
- Frédéric Loiret, Juan Navas, Jean-Philippe Babau et Olivier Lobry. *Component-Based Real-Time Operating System for Embedded Applications.* In Proceedings of the 12th International Symposium on Component-Based Software Engineering, CBSE '09, pages 209–226, Berlin, Heidelberg, 2009. Springer-Verlag.
- Olivier Lobry, Juan Navas et Jean-Philippe Babau. *Optimizing Component-Based Embedded Software.* Computer Software and Applications Conference, Annual International, vol. 2, pages 491–496, 2009.
- Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polajovic, Jacques Pulou, Marc Poulhies, Stephane Seyvoz, Julien Tous et Thomas Watteyne. *Think : View-Based Support of Non-Functional Properties in Embedded Systems.* In Proceedings of the 6th International Conference on Embedded Software and Systems (ICESS-09), 2009.

Bibliographie

[Abril Garcia 2005] Ana Belen Abril Garcia. *Estimation et optimisation de la consommation dans les descriptions architecturales des systèmes intégrés complexes*. PhD thesis, Université Paris VI, 2005. (Cité en page 37.)

[Adve 2003] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla et Brian Gaeke. *LLVA : A Low-level Virtual Instruction Set Architecture*. In MICRO 36 : Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, page 205, Washington, DC, USA, 2003. IEEE Computer Society. (Cité en page 46.)

[Agarwal 1991] Rakesh K. Agarwal. 80x86 architecture and programming (vol. ii) : architecture reference. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. (Cité en page 9.)

[Angelov 2005] Christo Angelov, Krzysztof Sierszecki1 et Nicolae Marian. *Component-Based Design of Embedded Software : An Analysis of Design Issues*. In Scientific Engineering of Distributed Java Applications : 4th International Workshop FIDJI. Springer, 2005. (Cité en page 151.)

[Angelov 2006] Christo Angelov, Xu Ke et Krzysztof Sierszecki. *A Component-Based Framework for Distributed Control Systems*. EUROMICRO Conference, vol. 0, pages 20–27, 2006. (Cité en page 19.)

[Anne 2009] Matthieu Anne, Ruan He, Tahar Jarboui, Marc Lacoste, Olivier Lobry, Guirec Lorant, Maxime Louvel, Juan Navas, Vincent Olive, Juraj Polajovic, Jacques Pulou, Marc Poulhies, Stephane Seyvoz, Julien Tous et Thomas Watteyne. *Think : View-Based Support of Non-Functional Properties in Embedded Systems*. In Proceedings of the 6th International Conference on Embedded Software and Systems (ICESS-09), 2009. (Cité en pages 21, 29, 62, 67 et 147.)

[AonixPERC 2010] AonixPERC. *The PERC Product Family*. http://www.aonix.com/pdf/PERC-Family_e.pdf, 2010. (Cité en pages 12 et 31.)

[ARM 2004] ARM. *ARM7TDMI Technical Reference Manual Rev. r4p1*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0210c/DDI0210B.pdf, 2004. (Cité en page 143.)

[ARM 2006] ARM. *CortexTM-M3 Technical Reference Manual Rev. r1p1*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337e/DDI0337E_cortex_m3_r1p1_trm.pdf, 2006. (Cité en page 149.)

[Arnold 2009] Jeff Arnold et M. Frans Kaashoek. *Ksplice : automatic rebootless kernel updates*. In EuroSys '09 : Proceedings of the 4th ACM European conference on Computer systems, pages 187–198, New York, NY, USA, 2009. ACM. (Cité en pages 60, 61 et 64.)

[Atmel 2010a] Atmel. *AT91SAM7S Series - Safe and Secure Bootloader Implementation*. http://www.atmel.com/dyn/resources/prod_documents/doc6282.pdf, 2010. (Cité en page 163.)

[Atmel 2010b] Atmel. *AT91SAM7S Series Preliminary.* http://www.atmel.com/dyn/resources/prod_documents/doc6175.pdf, 2010. (Cité en page 142.)

[Atmel 2010c] Atmel. *Atmel Corp.* http://www.atmel.com/, 2010. (Cité en page 142.)

[AVR 2010a] AVR. *8-bit Microcontroller with 4/8/16K Bytes In-System Programmable Flash - Reference.* http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf, 2010. (Cité en page 143.)

[AVR 2010b] AVR. *Atmel AVR MCU's.* http://www.atmel.com/products/AVR/, 2010. (Cité en page 143.)

[Azumi 2007] Takuya Azumi, Masanari Yamamoto, Yasuo Kominami, Nobuhisa Takagi, Hiroshi Oyama et Hiroaki Takada. *A New Specification of Software Components for Embedded Systems.* In ISORC '07 : Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pages 46–50, Washington, DC, USA, 2007. IEEE Computer Society. (Cité en page 19.)

[Babau 2007] Jean-Philippe Babau et Julien. DeAntoni. *Architectures logicielles pour les systèmes embarqués temps réel.* In Ecole d'été temps réel ETR'07, pages 75–94, Nantes, France, 2007. (Cité en pages 19 et 20.)

[Badea 2007] Carmen Badea, Alexandru Nicolau et Alexander V. Veidenbaum. *A simplified java bytecode compilation system for resource-constrained embedded processors.* In CASES '07 : Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems, pages 218–228, New York, NY, USA, 2007. ACM. (Cité en pages 38 et 41.)

[Badea 2008] Carmen Badea, Alexandru Nicolau et Alexander V. Veidenbaum. *Impact of JVM superoperators on energy consumption in resource-constrained embedded systems.* SIGPLAN Not., vol. 43, no. 7, pages 23–30, 2008. (Cité en page 38.)

[Baker 1999] Brenda S. Baker. *Parameterized diff.* In SODA '99 : Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms, pages 854–855, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. (Cité en page 60.)

[Balasubramanian 2008] Krishnakumar Balasubramanian et Douglas C. Schmidt. *Physical Assembly Mapper : A Model-Driven Optimization Tool for QoS-Enabled Component Middleware.* In RTAS '08 : Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, pages 123–134, Washington, DC, USA, 2008. IEEE Computer Society. (Cité en pages 38 et 41.)

[Barrett 2007] Steven F. Barrett, Daniel Pack et Mitchell Thornton. Atmel avr microcontroller primer : Programming and interfacing. Morgan & Claypool Publishers, 1st édition, 2007. (Cité en page 9.)

[Bass 1997] Len Bass, Paul Clements et Rick Kazman. Software Architecture in Practice. Addison-Wesley Professional, 1st édition, December 1997. (Cité en page 14.)

[Baumann 2005] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski et Jeremy Kerr. *Providing dynamic update in an operating system.* In ATEC '05 : Proceedings of the annual conference on USENIX Annual Technical Conference, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association. (Cité en page 56.)

[Baynes 2001] Kathleen Baynes, Chris Collins, Eric Fiterman, Brinda Ganesh, Paul Kohout, Christine Smit, Tiebing Zhang et Bruce Jacob. *The performance and energy consumption of three embedded real-time operating systems.* In CASES '01 : Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems, pages 203–210, New York, NY, USA, 2001. ACM. (Cité en page 38.)

[Becker 2006] Steffen Becker et Ralf Reussner. *The Impact of Software Component Adaptation on Quality of Service Properties.* L´objet, vol. 12, no. 1, pages 105–125, 2006. (Cité en page 39.)

[Becker 2008] Steffen Becker, Tobias Dencker et Jens Happe. *Model-Driven Generation of Performance Prototypes.* In Performance Evaluation : Metrics, Models and Benchmarks (SIPEW 2008), volume 5119 of *Lecture Notes in Computer Science*, pages 79–98. Springer-Verlag Berlin Heidelberg, 2008. (Cité en page 40.)

[Bennett 2000] Keith H. Bennett et Václav T. Rajlich. *Software maintenance and evolution : a roadmap.* In ICSE '00 : Proceedings of the Conference on The Future of Software Engineering, pages 73–87, New York, NY, USA, 2000. ACM. (Cité en page 50.)

[Beugnard 1999] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau et Damien Watkins. *Making Components Contract Aware.* Computer, vol. 32, no. 7, pages 38–45, 1999. (Cité en page 21.)

[Beugnard 2010] Antoine Beugnard, Jean-Marc Jezéquel et Noël Plouzeau. *Contract aware components, 10 years after.* Electronic proceedings in theoretical computer science, no. 37, pages 1 – 11, october 2010. (Cité en page 21.)

[Bomberger 1992] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau et Jonathan S. Shapiro. *The KeyKOS Nanokernel Architecture.* In Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association. (Cité en page 12.)

[Borde 2009] E. Borde, Gregory Haik et Laurent Pautet. *Mode-Based Reconfiguration of Critical Software Component Architectures.* In Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09., 20-24 2009. (Cité en pages 53, 61, 62 et 64.)

[Bouyssounouse 2005] Bruno Bouyssounouse et Joseph Sifakis. Embedded systems design : The artist roadmap for research and development (lecture notes in computer science). Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. (Cité en page 14.)

[Brouwers 2009] Niels Brouwers, Koen Langendoen et Peter Corke. *Darjeeling, a feature-rich VM for the resource poor.* In SenSys '09 : Proceedings of the 7th

ACM Conference on Embedded Networked Sensor Systems, pages 169–182, New York, NY, USA, 2009. ACM. (Cité en page 53.)

[Bruneton 2004a] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma et J.B. Stefani. *An Open Component Model and its Support in Java*. In Springer, editeur, Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 7), page 7 22, 2004. (Cité en pages 23 et 29.)

[Bruneton 2004b] Eric Bruneton, Thierry Coupaye et Jean-Bernard Stefani. *The Fractal Component Model*. http://fractal.objectweb.org/specification/, 2004. Version 2.0-3. (Cité en pages 29 et 82.)

[Buckley 2005] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid et Günter Kniesel. *Towards a taxonomy of software change : Research Articles*. J. Softw. Maint. Evol., vol. 17, no. 5, pages 309–332, 2005. (Cité en page 50.)

[Bunch 1996] S. Bunch, R. Hochsprung et T. Moore. *PowerPC Platform : a system architecture*. In Proceedings of the 41st IEEE International Computer Conference, COMPCON '96, pages 140–, Washington, DC, USA, 1996. IEEE Computer Society. (Cité en page 9.)

[Buonadonna 2001] Philip Buonadonna, Jason Hill et David Culler. *Active Message Communication for Tiny Networked Sensors*, 2001. (Cité en page 25.)

[Burke 2006] Bill Burke et Richard Monson-Haefel. Enterprise javabeans 3.0 (5th edition). O'Reilly Media, Inc., 2006. (Cité en pages 15 et 23.)

[Burns 1995] Alan Burns, Ken Tindell et Andy Wellings. *Effective Analysis for Engineering Real-Time Fixed Priority Schedulers*. IEEE Trans. Softw. Eng., vol. 21, pages 475–480, May 1995. (Cité en page 146.)

[C 2001] Ned C, Joanne E. Hale, Khaled Md. Kham, Juan F. Ramil et Wui-Gee Tan. *Types of software evolution and software maintenance*. Journal of Software Maintenance, vol. 13, no. 1, pages 3–30, 2001. (Cité en page 50.)

[Callou 2008] Gustavo Rau de Almeida Callou, Paulo Romero Martins Maciel, Ermeson Carneiro de Andrade, Bruno Costa e Silva Nogueira et aes Tavares Eduardo Antonio Guimar˙ *A coloured petri net based approach for estimating execution time and energy consumption in embedded systems*. In SBCCI '08 : Proceedings of the 21st annual symposium on Integrated circuits and system design, pages 134–139, New York, NY, USA, 2008. ACM. (Cité en page 38.)

[CAmkES 2010] CAmkES. *CAmkES web site*. http://ertos.nicta.com.au/research/camkes/, 2010. (Cité en pages 23 et 25.)

[Carzaniga 1998] Antonio Carzaniga, Alfonso Fuggetta, Rixhard S. Hall, Dennis Heimbigner, André van der Hoek et Alexander L. Wolf. *A Characterization Framework for Software Deployment Technologies*. Rapport technique, 1998. (Cité en pages 16 et 18.)

[CAS 2009] Centre d'analyse stratégique Premier Ministre Republique Française CAS. *La société et l'économie à l'aune de la révolution numérique*. http://www.strategie.gouv.fr/IMG/pdf/fiche_variable_1.2.6.pdf, 2009. Rapport de la Commission économie numérique présidée par Alain Bravo. (Cité en page 37.)

[Chambers 1989] C. Chambers et D. Ungar. *Customization : optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language.* In PLDI '89 : Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pages 146–160, New York, NY, USA, 1989. ACM. (Cité en pages 38 et 41.)

[Clausen 2000] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel et Gilles Muller. *Java bytecode compression for low-end embedded systems.* ACM Trans. Program. Lang. Syst., vol. 22, no. 3, pages 471–489, 2000. (Cité en page 41.)

[Corporation 1999] Atmel Corporation. *ARM7TDMI (Thumb) Datasheet*, 1999. (Cité en page 44.)

[Corporation 2010] Atmel Corporation. *FAQ : How many clock cycles does the ARM7TDMI take to execute ARM and Thumb instructions ?* `http://support.atmel.no`, 2010. (Cité en page 44.)

[Costa 2007] Paolo Costa, Geoff Coulson, Richard Gold, Manish Lad, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, Thirunavukkarasu Sivaharan, Nirmal Weerasinghe et Stefanos Zachariadis. *The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario.* In Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications, pages 69–78, Washington, DC, USA, 2007. IEEE Computer Society. (Cité en pages 19 et 31.)

[Coulson 2008] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama et Thirunavukkarasu Sivaharan. *A generic component model for building systems software.* ACM Trans. Comput. Syst., vol. 26, pages 1 :1–1 :42, March 2008. (Cité en page 19.)

[Crk 2009] Igor Crk, F Albinali, Chris Gniady et J Hartman. *Understanding Energy Consumption of Sensor Enabled Applications on Mobile Phones.* 2009. (Cité en page 37.)

[Crnkovic 2004] Ivica Crnkovic. *Component-based approach for embedded systems.* In In Proceedings of 9th International Workshop on Component-Oriented Programming, 2004. (Cité en page 151.)

[Crnkovic 2006] Ivica Crnkovic, Michel Chaudron et Stig Larsson. *Component-Based Development Process and Component Lifecycle.* Software Engineering Advances, International Conference on, vol. 0, page 44, 2006. (Cité en pages 15 et 16.)

[CSR 2010] CSR. *BlueCore Bluetooth Solutions.* `http://www.csr.com/products/technology/bluetooth`, 2010. (Cité en page 145.)

[Cuenot 2007] Philippe Cuenot, DeJiu Chen, Sebastien Gerard, Henrik Lonn, Mark-Oliver Reiser, David Servat, Carl-Johan Sjostedt, Ramin Tavakoli Kolagari, Martin Torngren et Matthias Weber. *Managing Complexity of Automotive Electronics Using the EAST-ADL.* In Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems, pages 353–358, Washington, DC, USA, 2007. IEEE Computer Society. (Cité en page 19.)

[De Bus 2003] Bruno De Bus, Daniel Kästner, Dominique Chanet, Ludo Van Put et Bjorn De Sutter. *Post-pass compaction techniques.* Commun. ACM, vol. 46, no. 8, pages 41–46, 2003. (Cité en page 41.)

[De Sutter 2005] Bjorn De Sutter, Bruno De Bus et Koen De Bosschere. *Link-time binary rewriting techniques for program compaction.* ACM Trans. Program. Lang. Syst., vol. 27, no. 5, pages 882–945, 2005. (Cité en page 41.)

[Dea 2007] Software deployment, past, present and future, 2007. (Cité en page 16.)

[DeAntoni 2005] Julien DeAntoni et Jean-Philippe Babau. *A MDA-based approach for real time embedded systems simulation.* Distributed Simulation and Real-Time Applications, IEEE International Symposium on, vol. 0, pages 257–264, 2005. (Cité en pages 13 et 151.)

[DeRemer 1975] Frank DeRemer et Hans Kron. *Programming-in-the large versus programming-in-the-small.* In Proceedings of the international conference on Reliable software, pages 114–121, New York, NY, USA, 1975. ACM. (Cité en pages 13 et 14.)

[Duesterwald 2005] E. Duesterwald. *Design and Engineering of a Dynamic Binary Optimizer.* Proceedings of the IEEE, vol. 93, no. 2, pages 436 –448, feb. 2005. (Cité en pages 38 et 41.)

[Dunkels 2004] Adam Dunkels, Bjorn Gronvall et Thiemo Voigt. *Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors.* Local Computer Networks, Annual IEEE Conference on, vol. 0, pages 455–462, 2004. (Cité en page 57.)

[Dunkels 2006] Adam Dunkels, Niclas Finne, Joakim Eriksson et Thiemo Voigt. *Run-time dynamic linking for reprogramming wireless sensor networks.* In SenSys '06 : Proceedings of the 4th international conference on Embedded networked sensor systems, pages 15–28, New York, NY, USA, 2006. ACM. (Cité en pages 57 et 64.)

[EE Times Group 2010] EE Times Group et UBM Media. *2010 Embedded Market Study.* http://www.eetimes.com/electrical-engineers/education-training/webinars/4006580/2010-Embedded-Market-Study, 2010. (Cité en page 12.)

[ELF 1995] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification.* http ://x86.ddj.com/ftp/manuals/tools/elf.pdf, May 1995. (Cité en page 57.)

[Engler 1995] D. R. Engler, M. F. Kaashoek et J. O'Toole Jr. *Exokernel : an operating system architecture for application-level resource management.* In Proceedings of the fifteenth ACM symposium on Operating systems principles, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM. (Cité en page 12.)

[Escoffier 2007] Clement Escoffier et Richard S. Hall. *Dynamically adaptable applications with iPOJO service components.* In SC'07 : Proceedings of the 6th international conference on Software composition, pages 113–128, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité en page 23.)

[Fassino 2002] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall et Gilles Muller. *Think : A Software Framework for Component-based Operating System Kernels.* In ATEC '02 : Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, pages 73–86, Berkeley, CA, USA, 2002. USENIX Association. (Cité en pages 19, 21 et 29.)

[Feiler 2004] P H. Feiler, B. Lewis, S. Vestal et E. Colbert. *An overview of the SAE Architecture & Design Language (AADL) Standard : A Basis for Model-Based Architecture-Driven Embedded Systems Engineering.* In Architecture Description Language, workshop at IFIP World Computer Congress, 2004. (Cité en pages 19 et 53.)

[Ford 1997] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin et Olin Shivers. *The Flux OSKit : a substrate for kernel and language research.* SIGOPS Oper. Syst. Rev., vol. 31, pages 38–51, October 1997. (Cité en page 12.)

[Forsberg 1991] Kevin Forsberg et Harold Mooz. *The Relationship of System Engineering to the Project Cycle.* In Proceedings of the 12th INTERNET World Congress on Project Management, 1991. (Cité en page 16.)

[Fractal 2010] Fractal. *Fractal Project.* `http://fractal.objectweb.org`, 2010. (Cité en pages 23 et 29.)

[FSF 2010a] Free Software Foundation Inc FSF. *The GNU Compiler Collection - Options That Control Optimization.* `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, 2010. (Cité en page 43.)

[FSF 2010b] Free Software Foundation Inc FSF. *The GNU Compiler Collection Manual Version 4.5.* `http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gcc/`, 2010. (Cité en pages 24, 40 et 42.)

[Fu, Luotao and Schwebel, Robert 2010] Fu, Luotao and Schwebel, Robert. *RT Preempt How To.* `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`, 2010. (Cité en page 156.)

[Gamma 1995] Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides. Design patterns : elements of reusable object-oriented software. Addison-Wesley Professional, 1995. (Cité en page 151.)

[Gar 2003] Formal modeling and analysis of software architecture : Components, connectors, and events, volume 2804, pages 1–24. Springer, 2003. (Cité en page 14.)

[Garlan 2003] David Garlan, Shang-Wen Cheng et Bradley Schmerl. *Architecting dependable systems.* chapitre Increasing system dependability through architecture-based self-repair, pages 61–89. Springer-Verlag, Berlin, Heidelberg, 2003. (Cité en page 14.)

[Gay 2003] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer et David Culler. *The nesC language : A holistic approach to networked embedded systems.* In PLDI '03 : Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 1–11, New York, NY, USA, 2003. ACM. (Cité en page 24.)

[Gay 2005] David Gay, Phil Levis et David Culler. *Software design patterns for TinyOS*. In LCTES '05 : Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pages 40–49, New York, NY, USA, 2005. ACM. (Cité en page 151.)

[Genssler 2002] 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 Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, CASES '02, pages 19–26, New York, NY, USA, 2002. ACM. (Cité en page 19.)

[Gerber 2002] Richard Gerber. The software optimization cookbook. Intel Press, 2002. (Cité en page 40.)

[Gosling 2000] James Gosling, Bill Joy, Guy Steele et Gilad Bracha. Java language specification, second edition : The java series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2000. (Cité en page 22.)

[Gössler 2007] Gregor Gössler, Sussane Graf, Mila Majster-Cederbaum, M. Martens et Joseph Sifakis. *An Approach to Modelling and Verification of Component Based Systems*. In Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science, SOFSEM '07, pages 295–308, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité en page 14.)

[Grace 2006] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter et Danny Hughes. *Dynamic reconfiguration in sensor middleware*. In MidSens '06 : Proceedings of the international workshop on Middleware for sensor networks, pages 1–6, New York, NY, USA, 2006. ACM. (Cité en page 54.)

[Grone 2005] Bernhard Grone, Andreas Knopfel et Peter Tabeling. *Component vs. Component : Why We Need More Than One Definition*. In ECBS '05 : Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems, pages 550–552, Washington, DC, USA, 2005. IEEE Computer Society. (Cité en page 15.)

[Guo 2008] Yu Guo, Krzysztof Sierszecki et Christo Angelov. *A (Re)Configuration Mechanism for Resource-Constrained Embedded Systems*. In COMPSAC '08 : Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference, pages 1315–1320, Washington, DC, USA, 2008. IEEE Computer Society. (Cité en pages 59 et 64.)

[Han 2005] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler et Mani Srivastava. *A dynamic operating system for sensor nodes*. In MobiSys '05 : Proceedings of the 3rd international conference on Mobile systems, applications, and services, pages 163–176, New York, NY, USA, 2005. ACM. (Cité en page 58.)

[Hansson 2004] Hans Hansson, Mikael Akerholm, Ivica Crnkovic et Martin Torngren. *SaveCCM - A Component Model for Safety-Critical Real-Time Systems*. In Proceedings of the 30th EUROMICRO Conference, pages 627–635, Washington, DC, USA, 2004. IEEE Computer Society. (Cité en page 19.)

[Heiser 2005] Gernot Heiser. *Secure Embedded Systems need microkernels.* USENIX, vol. 30, no. 6, pages 9–13, 2005. (Cité en page 26.)

[Helander 1998] Johannes Helander et Alessandro Forin. *MMLite : a highly componentized system architecture.* In EW 8 : Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications, pages 96–103, New York, NY, USA, 1998. ACM. (Cité en pages 19, 56 et 171.)

[Hellerstein 2004] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh et Dawn M. Tilbury. Feedback control of computing systems. John Wiley & Sons, 2004. (Cité en page 51.)

[Henderson 2006] Bryan Henderson. *Linux Loadable Kernel Module HOWTO.* `http://tldp.org/HOWTO/Module-HOWTO/`, 2006. (Cité en page 57.)

[Henzinger 2006] Thomas A. Henzinger et Joseph Sifakis. *The Embedded Systems Design Challenge.* In Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science, pages 1–15. Springer, 2006. (Cité en pages 7 et 35.)

[Hill 2002] Jason L. Hill et David E. Culler. *Mica : A Wireless Platform for Deeply Embedded Networks.* IEEE Micro, vol. 22, pages 12–24, 2002. (Cité en page 25.)

[Hong 2002] Jinpyo Hong. *Memory optimization techniques for embedded systems.* PhD thesis, 2002. Director-Ramanujam, Jagannathan. (Cité en page 40.)

[Horowitz 2003] Benjamin Horowitz. *Giotto : a time-triggered language for embedded programming.* PhD thesis, 2003. Chair-Henzinger, Thomas A. (Cité en pages 19 et 21.)

[Howell 1999] Jon Howell et Mark Montague. *Hey, You Got Your Compiler in My Operating System !* In HOTOS '99 : Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems, page 122, Washington, DC, USA, 1999. IEEE Computer Society. (Cité en page 41.)

[Hsieh 2004] Paul Hsieh. *Programming Optimization.* `http://www.azillionmonkeys.com/qed/optimize.html`, 2004. (Cité en page 40.)

[Hughes 2009] Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens et Wouter Joosen. *LooCI : a loosely-coupled component infrastructure for networked embedded systems.* In MoMM '09 : Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia, pages 195–203, New York, NY, USA, 2009. ACM. (Cité en pages 19 et 54.)

[Hugues 2007] Jérôme Hugues, Laurent Pautet et Bechir Zalila. *From MDD to Full Industrial Process : Building Distributed Real-Time Embedded Systems for the High-Integrity Domain.* In Fabrice Kordon et Oleg Sokolsky, editeurs, Composition of Embedded Systems. Scientific and Industrial Issues, volume 4888 of *Lecture Notes in Computer Science*, pages 35–52. Springer Berlin / Heidelberg, 2007. (Cité en page 19.)

[Hui 2004] Jonathan W. Hui et David Culler. *The dynamic behavior of a data dissemination protocol for network programming at scale.* In SenSys '04 : Proceedings of the 2nd international conference on Embedded networked sensor systems, pages 81–94, New York, NY, USA, 2004. ACM. (Cité en page 57.)

[iABG 1992] iABG. *V-Model Lifecycle Process Model Brief Description.* http://www.v-modell.iabg.de/kurzb/vm/k_vm_e.doc, 1992. (Cité en page 16.)

[IAR 2010] IAR. *Embedded Development Tools from IAR Systems.* http://www.iar.com, 2010. (Cité en pages 30 et 143.)

[IEEE 1998] IEEE et IEA. *Industry Implementation of Internacional Standard ISO/IEC 12207 :1995.* IEEE/EIA Guide, April 1998. (Cité en page 16.)

[Institute 2002] British Standards Institute. The c standard. John Wiley & Sons, Inc., New York, NY, USA, 2002. (Cité en page 12.)

[Isovic 2002] Damir Isovic et Christer Norstrm. *Components in Real-Time Systems.* In Proc. of the 8th International Conference on Real-Time Computing Systems and Applications, pages 135–139, 2002. (Cité en pages 19 et 21.)

[Jeong 2004] Jaein Jeong. *Incremental network programming for wireless sensors.* In Proceedings of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks IEEE SECON, pages 25–33, 2004. (Cité en pages 60 et 61.)

[John 2010] Karl Heinz John et Michael Tiegelkamp. Iec 61131-3 : Programming industrial automation systems concepts and programming languages, requirements for programming systems, decision-making aids. Springer Publishing Company, Incorporated, 2nd édition, 2010. (Cité en page 59.)

[Jones 1996] Neil D. Jones. *An introduction to partial evaluation.* ACM Comput. Surv., vol. 28, no. 3, pages 480–503, 1996. (Cité en page 41.)

[Kandemir 2005] Mahmut Kandemir, Feihui Li, Guilin Chen, Guangyu Chen et Ozcan Ozturk. *Studying Storage-Recomputation Tradeoffs in Memory-Constrained Embedded Processing.* In DATE '05 : Proceedings of the conference on Design, Automation and Test in Europe, pages 1026–1031, Washington, DC, USA, 2005. IEEE Computer Society. (Cité en pages 36 et 38.)

[Kane 1988] Gerry Kane. Mips risc architecture. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. (Cité en page 9.)

[Ke 2007] Xu Ke, Krzysztof Sierszecki et Christo Angelov. *COMDES-II : A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems.* In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA '07, pages 199–208, Washington, DC, USA, 2007. IEEE Computer Society. (Cité en page 19.)

[Keil 2009] Keil. *STM32F103xx advanced ARM-based 32-bit MCU Reference manual.* http://www.keil.com/dd/docs/datashts/st/stm32f10xxx.pdf, 2009. (Cité en page 149.)

[Keil 2010] Keil. *MCBSTM32 Evaluation Board.* http://www.keil.com/mcbstm32/, 2010. (Cité en page 149.)

[Kemerer 1999] Chris F. Kemerer et Sandra Slaughter. *An Empirical Approach to Studying Software Evolution.* IEEE Trans. Softw. Eng., vol. 25, no. 4, pages 493–509, 1999. (Cité en page 50.)

[Kernighan 1988] Brian W. Kernighan. The c programming language. Prentice Hall Professional Technical Reference, 2nd édition, 1988. (Cité en page 12.)

[Kilby 1964] Jack S. Kilby. *Miniaturized electronic circuits*, June 1964. (Cité en page 14.)

[Knuth 1979] D. Knuth. Structured programming with go to statements, pages 257–321. Yourdon Press, Upper Saddle River, NJ, USA, 1979. (Cité en page 141.)

[Koshy 2005] Joel Koshy et Raju Pandey. *VMSTAR : synthesizing scalable runtime environments for sensor networks.* In SenSys '05 : Proceedings of the 3rd international conference on Embedded networked sensor systems, pages 243–254, New York, NY, USA, 2005. ACM. (Cité en pages 53, 60 et 61.)

[Koshy 2008] Joel Koshy, Ingwar Wirjawan, Raju Pandey et Yann Ramin. *Balancing computation and communication costs : The case for hybrid execution in sensor networks.* Ad Hoc Netw., vol. 6, no. 8, pages 1185–1200, 2008. (Cité en page 53.)

[Koshy 2009] Joel Koshy, Raju Pandey et Ingwar Wirjawan. *Optimizing Embedded Virtual Machines.* In CSE '09 : Proceedings of the 2009 International Conference on Computational Science and Engineering, pages 342–351, Washington, DC, USA, 2009. IEEE Computer Society. (Cité en page 53.)

[Koziolek 2010] Heiko Koziolek. *Performance evaluation of component-based software systems : A survey.* Perform. Eval., vol. 67, no. 8, pages 634–658, 2010. (Cité en page 38.)

[Krasner 2008] Jerry Krasner. *A Model Driven Approach to Software Development for Systems*, 2008. (Cité en page 14.)

[Krieger 2006] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland et Volkmar Uhlig. *K42 : building a complete operating system.* In Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM. (Cité en page 12.)

[Ksplice 2010] Ksplice. *Ksplice web site.* http://www.ksplice.com/, 2010. (Cité en page 60.)

[Kuz 2007a] Ihor Kuz et Yan Liu. *Extending the capabilities of component models for embedded systems.* In QoSA'07 : Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications, pages 182–196, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité en page 55.)

[Kuz 2007b] Ihor Kuz, Yan Liu, Ian Gorton et Gernot Heiser. *CAmkES : A Component Model for Secure Microkernel-based Embedded Systems.* J. Syst. Softw., vol. 80, no. 5, pages 687–699, 2007. (Cité en pages 19, 21, 23 et 25.)

[Kwong 2010] C.K. Kwong, L.F. Mu, J.F. Tang et X.G. Luo. *Optimization of software components selection for component-based software system development.* Computers & Industrial Engineering, vol. 58, no. 4, pages 618 – 624, 2010. (Cité en page 40.)

[Labrosse 2002] Jean J. Labrosse. Microc/os-ii : the real-time kernel. CMP Media, Inc., USA, 2nd édition, 2002. (Cité en pages 145 et 146.)

[Larman 2003] Craig Larman et Victor R. Basili. *Iterative and Incremental Development : A Brief History.* Computer, vol. 36, pages 47–56, 2003. (Cité en page 16.)

[Lattner 2004] Chris Lattner et Vikram Adve. *LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation.* In CGO '04 : Proceedings of the international symposium on Code generation and optimization, page 75, Washington, DC, USA, 2004. IEEE Computer Society. (Cité en page 46.)

[Lau 2007] K.-K. Lau et Z. Wang. *Software Component Models.* IEEE Trans. on Software Engineering, vol. 33, no. 10, pages 709–724, October 2007. (Cité en page 16.)

[Lee 2001] Sheayun Lee, Andreas Ermedahl et Sang Lyul Min. *An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors.* In LCTES '01 : Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, pages 1–10, New York, NY, USA, 2001. ACM. (Cité en page 41.)

[Lee 2006] Edward A. Lee. *Cyber-Physical Systems - Are Computing Foundations Adequate ?* In Position Paper for NSF Workshop On Cyber-Physical Systems : Research Motivation, Techniques and Roadmap, 2006. (Cité en page 8.)

[Lee 2008] Edward A. Lee. *Cyber Physical Systems : Design Challenges.* In ISORC '08 : Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, pages 363–369, Washington, DC, USA, 2008. IEEE Computer Society. (Cité en page 8.)

[Leger 2009] Marc Leger. *Fiabilité des Reconfigurations Dynamiques dans les Architectures à Composants.* PhD thesis, Ecole Nationale Superieure des Mines de Paris, 2009. (Cité en page 56.)

[Lego 2010] Lego. *The Lego Mindstorms NXT Homepage.* `http://mindstorms.lego.com/`, 2010. (Cité en page 142.)

[Lehman 1985] M. M. Lehman et L. A. Belady, editeurs. Program evolution : processes of software change. Academic Press Professional, Inc., San Diego, CA, USA, 1985. (Cité en pages 1, 59 et 60.)

[Lehman 2001] M. M. Lehman et J. F. Ramil. *Evolution in software and related areas.* In IWPSE '01 : Proceedings of the 4th International Workshop on

Principles of Software Evolution, pages 1–16, New York, NY, USA, 2001. ACM. (Cité en page 50.)

[LejOS 2010a] LejOS. *LejOS - Java for Lego Mindstorms.* http://lejos.sourceforge.net/, 2010. (Cité en page 143.)

[LejOS 2010b] LejOS. *LejOS : Java for Lego Mindstorms.* http://lejos.sourceforge.net/, 2010. (Cité en page 12.)

[Lestideau 2002] Vincent Lestideau, Noureddine Belkhatir et Pierre yves Cunin. *Towards Automated Software Component configuration and deployment.* 2002. (Cité en page 16.)

[Leupers 1999] Rainer Leupers et Peter Marwedel. *Function inlining under code size constraints for embedded processors.* In ICCAD '99 : Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design, pages 253–256, Piscataway, NJ, USA, 1999. IEEE Press. (Cité en page 38.)

[Levis 2002] Philip Levis et David Culler. *Maté : A Tiny Virtual Machine for Sensor Networks.* In ASPLOS-X : Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, pages 85–95, New York, NY, USA, 2002. ACM. (Cité en pages 31 et 53.)

[Levis 2005] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer et D. Culler. *TinyOS : An Operating System for Sensor Networks.* In Werner Weber, Jan M. Rabaey et Emile Aarts, editeurs, Ambient Intelligence, pages 115–148. Springer Berlin Heidelberg, 2005. (Cité en pages 19 et 23.)

[Liedtke 1995] J. Liedtke. *On Micro-Kernel Construction.* In SOSP '95 : Proceedings of the fifteenth ACM symposium on Operating systems principles, pages 237–250, New York, NY, USA, 1995. ACM. (Cité en page 26.)

[Lobry 2008] Olivier Lobry et Juraj Polakovic. *Controlling the Performance Overhead of Component-Based Systems.* In Cesare Pautasso et Éric Tanter, editeurs, Software Composition, volume 4954 of *Lecture Notes in Computer Science*, pages 149–156. Springer Berlin / Heidelberg, 2008. (Cité en page 147.)

[Lobry 2009] Olivier Lobry, Juan Navas et Jean-Philippe Babau. *Optimizing Component-Based Embedded Software.* Computer Software and Applications Conference, Annual International, vol. 2, pages 491–496, 2009. (Cité en pages 4 et 147.)

[Loiret 2009] Frédéric Loiret, Juan Navas, Jean-Philippe Babau et Olivier Lobry. *Component-Based Real-Time Operating System for Embedded Applications.* In Proceedings of the 12th International Symposium on Component-Based Software Engineering, CBSE '09, pages 209–226, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité en pages 4, 146, 155 et 159.)

[Loiret 2010] Frédéric Loiret, Lionel Seinturier, Laurence Duchien et David Servat. *A Three-Tier Approach for Composition of Real-Time Embedded Software Stacks.* In Lars Grunske, Ralf Reussner et Frantisek Plasil, editeurs, Component-Based Software Engineering, volume 6092 of *Lecture Notes in*

Computer Science, pages 37–54. Springer Berlin / Heidelberg, 2010. (Cité en page 83.)

[Matthys 2009] Nelson Matthys, Danny Hughes, Sam Michiels, Christophe Huygens et Wouter Joosen. *Fine-Grained Tailoring of Component Behaviour for Embedded Systems*. In SEUS '09 : Proceedings of the 7th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems, pages 156–167, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité en pages 54, 55 et 62.)

[Mazidi 2009] Muhammad Ali Mazidi, Rolin McKinlay et Danny Causey. Pic microcontroller and embedded systems. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd édition, 2009. (Cité en page 9.)

[McIlroy 1968] M. D. McIlroy. *Mass-produced software components*. Proc. NATO Conf. on Software Engineering, Garmisch, Germany, 1968. (Cité en page 14.)

[Medvidovic 2000] Nenad Medvidovic et Richard N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. IEEE Trans. Softw. Eng., vol. 26, no. 1, pages 70–93, 2000. (Cité en page 22.)

[Mehta 2000] Nikunj R. Mehta, Nenad Medvidovic et Sandeep Phadke. *Towards a taxonomy of software connectors*. In ICSE '00 : Proceedings of the 22nd international conference on Software engineering, pages 178–187, New York, NY, USA, 2000. ACM. (Cité en page 21.)

[Michiels 2006] Sam Michiels, Wouter Horré, Wouter Joosen et Pierre Verbaeten. *DAViM : a dynamically adaptable virtual machine for sensor networks*. In MidSens '06 : Proceedings of the international workshop on Middleware for sensor networks, pages 7–12, New York, NY, USA, 2006. ACM. (Cité en pages 31 et 53.)

[Micrium 2010] Micrium. *μC/OS-II - The Real-Time Kernel*. `http://micrium.com/page/products/rtos/os-ii`, 2010. (Cité en pages 145 et 146.)

[Microsoft 1999] Microsoft. *Component Object Model (COM)*. `http://msdn.microsoft.com`, Janvier 1999. (Cité en page 15.)

[Microsystems 2010] ORACLE Sun Microsystems. *Java Card 3.0 Platform Specification*. `http://java.sun.com/javacard/3.0/specs.jsp`, 2010. (Cité en page 12.)

[Moteiv 2006] Moteiv. *Tmote Sky Datasheet http ://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf*, 2006. (Cité en page 25.)

[Motorola 1989] CORPORATE Motorola Inc. M68000 family programmer's reference manual. Motorola Inc., Phoenix, AZ, USA, 1989. (Cité en page 9.)

[Mottola 2008] Luca Mottola, Gian Pietro Picco et Adil Amjad Sheikh. *FiGaRo : fine-grained software reconfiguration for wireless sensor networks*. In EWSN'08 : Proceedings of the 5th European conference on Wireless sensor networks, pages 286–304, Berlin, Heidelberg, 2008. Springer-Verlag. (Cité en page 58.)

[Müller 2007] René Müller, Gustavo Alonso et Donald Kossmann. *A virtual machine for sensor networks*. In EuroSys '07 : Proceedings of the 2nd ACM

SIGOPS/EuroSys European Conference on Computer Systems 2007, pages 145–158, New York, NY, USA, 2007. ACM. (Cité en page 53.)

[Munawar 2010] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel et Klaus Wehrle. *Dynamic TinyOS : Modular and Transparent Incremental Code-Updates for Sensor Networks*. In Proceedings of the IEEE International Conference on Communications (ICC), Cape Town, South Africa, May 23-27, 2010. (Cité en pages 58, 61 et 64.)

[Naik 2001] Kshirasagar Naik et David S. L. Wei. *Software implementation strategies for power-conscious systems*. Mob. Netw. Appl., vol. 6, no. 3, pages 291–305, 2001. (Cité en page 38.)

[Navas 2009a] Juan Navas et Jean-Philippe Babau. *Efficient and Adapted Component-Based Strategies for Embedded Software Device Drivers Development*. In Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 02, CSE '09, pages 514–519, Washington, DC, USA, 2009. IEEE Computer Society. (Cité en pages 4 et 151.)

[Navas 2009b] Juan F. Navas, Jean-Philippe Babau et Olivier Lobry. *Minimal yet effective reconfiguration infrastructures in component-based embedded systems*. In Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime (SINTER'09), pages 41–48, New York, NY, USA, 2009. ACM. (Cité en pages 4, 46, 56, 120 et 160.)

[Navas 2010a] Juan F. Navas, Jean-Philippe Babau et Jacques Pulou. *A component-based run-time evolution infrastructure for resource-constrained embedded systems*. In Proceedings of the ninth international conference on Generative programming and component engineering, GPCE '10, pages 73–82, New York, NY, USA, 2010. ACM. (Cité en pages 4 et 163.)

[Navas 2010b] Juan F. Navas, Jean-Philippe Babau et Jacques Pulou. *An Optimized Run-time Evolution Infrastructure for Component-Based Embedded Systems*. In Proceedings of the ninth BElgian-NEtherlands software eVOLution seminar, BENEVOL '10, 2010. (Cité en page 4.)

[nesC 2010] nesC. *nesC Internals*. `http://docs.tinyos.net/index.php/Nesc-internals`, 2010. (Cité en page 24.)

[nxOS 2010] nxOS. *An embedded operating system toolkit for the Lego Mindstorms NXT*. `https://github.com/danderson/nxos/`, 2010. (Cité en pages 143 et 145.)

[NXTGCC 2010] NXTGCC. *The First Open Source LEGO MINDSTORMS NXT Firmware Development Kit*. `http://nxtgcc.sourceforge.net/`, 2010. (Cité en page 143.)

[NXTOSEK 2010] NXTOSEK. *ANSI C/C++ with OSEK/μITRON RTOS for LEGO MINDSTORMS NXT*. `http://lejos-osek.sourceforge.net/`, 2010. (Cité en page 143.)

[OMG 2003] OMG. *MDA Guide Version 1.0.1*. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`, June 2003. (Cité en page 13.)

[OMG 2010] OMG. *The official CORBA standard from the OMG group*. `http://www.omg.org/spec/CORBA/3.1/`, 2010. (Cité en pages 15 et 25.)

[Opt 2008] *Optimisation de Systèmes Embarqués - Episodes 1 au 4*. http://embedded.over-blog.com/article-19724092.html, 2008. (Cité en page 40.)

[Oreizy 1998] P. Oreizy et R. Taylor. *On the Role of Software Architectures in Runtime System Reconfiguration*. In CDS '98 : Proceedings of the International Conference on Configurable Distributed Systems, page 61, Washington, DC, USA, 1998. IEEE Computer Society. (Cité en pages 14, 51, 64 et 66.)

[Oreizy 2008] Peyman Oreizy, Nenad Medvidovic et Richard N. Taylor. *Runtime software adaptation : framework, approaches, and styles*. In ICSE Companion '08 : Companion of the 30th international conference on Software engineering, pages 899–910, New York, NY, USA, 2008. ACM. (Cité en pages 14, 51 et 64.)

[Panda 2001] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle et P. G. Kjeldsberg. *Data and memory optimization techniques for embedded systems*. ACM Trans. Des. Autom. Electron. Syst., vol. 6, no. 2, pages 149–206, 2001. (Cité en page 40.)

[Panta 2009] Rajesh Krishna Panta, Saurabh Bagchi et Samuel P. Midkiff. *Zephyr : Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation*. In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX), San Diego, CA, USA, June 14-19, 2009. (Cité en pages 60, 61 et 64.)

[Pedersen 2007] Rasmus Ulslev Pedersen. Tinyos education with lego mindstorms nxt, chapitre 14, pages 231–241. Springer Berlin Heidelberg, September 2007. (Cité en page 143.)

[Per 2002] Performance solutions : a practical guide to creating responsive, scalable software. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002. (Cité en page 40.)

[Percival 2003] Colin Percival. *Naive Differences of Executable Code*, 2003. (Cité en page 60.)

[Petazzoni 2009] Maxime Petazzoni. *TR54 : Présentation de NxOS*. http://blog.bulix.org/pub/2009/11/09/cours-nxos.pdf, 2009. (Cité en page 148.)

[Pilone 2005] Dan Pilone et Neil Pitman. Uml 2.0 in a nutshell (in a nutshell (o'reilly)). O'Reilly Media, Inc., 2005. (Cité en page 21.)

[Plásil 1998] F. Plásil, D. Bálek et R. Janecek. *SOFA/DCUP : Architecture for Component Trading and Dynamic Updating*. In Proceedings of the International Conference on Configurable Distributed Systems, CDS '98, pages 43–, Washington, DC, USA, 1998. IEEE Computer Society. (Cité en page 19.)

[Polakovic 2006] J. Polakovic, A. E. Ozcan et J.-B. Stefani. *Building Reconfigurable Component-Based OS with THINK*. In EUROMICRO Conference on Software Engineering and Advanced Applications, 2006. (Cité en pages 56, 61, 64 et 171.)

[Polakovic 2007] J. Polakovic, S. Mazare, J.-B. Stefani et P.-C. David. *Experience with Implementing Safe Reconfigurations in Component-Based Embedded Systems*. In International ACM Symposium on Component-Based Software Engineering (CBSE), 2007. (Cité en pages 56 et 83.)

[Polakovic 2008] Juraj Polakovic. *Architecture logicielle et outils pour systËmes d'exploitation reconfigurables.* PhD thesis, Institut National Polutechnique de Grenoble, 2008. (Cité en pages 29, 50, 56, 75 et 119.)

[Porter 2010] Barry Porter, Utz Roedig, Francois Taiani et Geoff Coulson. *A comparison of static and dynamic component models for Wireless Sensor Networks.* In Proceedings of the The First International Workshop on Networks of Cooperating Objects (CONET2010), Stockholm, Sweden, Avril 2010. (Cité en page 46.)

[Potier 2010] Dominique Potier. *Briques génériques du logiciel embarqué.* http://www.ladocumentationfrancaise.fr/rapports-publics/104000528/, 2010. Mission confiée par le Ministère de l'industrie, le Secrétariat d'Etat à la prospective et au développement de l'économie numérique, et le Commissariat général aux investissements d'avenir. (Cité en page 1.)

[Poulhies 2010] Marc Poulhies. *Conception et Implantation de Système Fondée sur les Composants. Vers une Unifcation des Paradigmes Géenie Logiciel et Système.* PhD thesis, Université de Grenoble, 2010. (Cité en pages 14, 29, 75 et 83.)

[Python 2010] Python. *The Python Language Reference.* http://docs.python.org/reference/, 2010. (Cité en page 26.)

[Reijers 2003] Niels Reijers et Koen Langendoen. *Efficient code distribution in wireless sensor networks.* In WSNA '03 : Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, pages 60–67, New York, NY, USA, 2003. ACM. (Cité en pages 60 et 61.)

[Reussner 2003] Ralf H. Reussner, Iman H. Poernomo et Heinz W. Schmidt. *Reasoning on Software Architectures with Contractually Specified Components.* In A. Cechich, M. Piattini et A. Vallecillo, editeurs, Component-Based Software Quality : Methods and Techniques, volume 2693 of *Lecture Notes in Computer Science*, pages 287–325. Springer-Verlag Berlin Heidelberg, 2003. (Cité en page 41.)

[Ritchie 1986] D M Ritchie et K Thompson. The unix time-sharing system, pages 1–25. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. (Cité en page 12.)

[Royce 1987] W. W. Royce. *Managing the development of large software systems : concepts and techniques.* In ICSE '87 : Proceedings of the 9th international conference on Software Engineering, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. (Cité en page 16.)

[Rozier 1991] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. LÃ©onard et W. Neuhauser. *Overview of the CHORUS Distributed Operating Systems.* Computing Systems, vol. 1, pages 39–69, 1991. (Cité en page 12.)

[Russello 2008] Giovanni Russello, Leonardo Mostarda et Naranker Dulay. *ESCAPE : A Component-Based Policy Framework for Sense and React Applications.* In CBSE '08 : Proceedings of the 11th International Symposium on

Component-Based Software Engineering, pages 212–229, Berlin, Heidelberg, 2008. Springer-Verlag. (Cité en page 54.)

[Schultz 2003] Ulrik Pagh Schultz, Kim Burgaard, Flemming Gram Christensen et Jørgen Lindskov Knudsen. *Compiling java for low-end embedded systems.* In LCTES '03 : Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, pages 42–50, New York, NY, USA, 2003. ACM. (Cité en page 41.)

[Seal 2000] David Seal. Arm architecture reference manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2000. (Cité en page 9.)

[Sentilles 2009] Séverine Sentilles, Petr Štěpán, Jan Carlson et Ivica Crnković. *Integration of Extra-Functional Properties in Component Models.* In CBSE '09 : Proceedings of the 12th International Symposium on Component-Based Software Engineering, pages 173–190, Berlin, Heidelberg, 2009. Springer-Verlag. (Cité en page 20.)

[Shaw 1996a] Mary Shaw. *Truth vs Knowledge : The Difference Between What a Component Does and What We Know It Does.* In IWSSD '96 : Proceedings of the 8th International Workshop on Software Specification and Design, page 181, Washington, DC, USA, 1996. IEEE Computer Society. (Cité en page 20.)

[Shaw 1996b] Mary Shaw et David Garlan. Software architecture : perspectives on an emerging discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. (Cité en page 22.)

[Shen 2007] Bor-Yeh Shen et Mei-Ling Chiang. *A server-side pre-linking mechanism for updating embedded clients dynamically.* In EUC'07 : Proceedings of the 2007 international conference on Embedded and ubiquitous computing, pages 146–157, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité en page 59.)

[Singh 1995] Raghu Singh. *International Standard ISO/IEC 12207 Software Life Cycle Processes.* Rapport technique, Federal Aviation Administration. Washington, DC, USA, 1995. (Cité en page 16.)

[Soules 2003] C. Soules, J. Appavoo, K. Hui, R.W. Wisniewski, D. DaSilva, G. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg et J. Xenidis. *System support for online reconfiguration.* In Proceedings of the 2003 USENIX Annual Technical Conference, pages 141–154, 2003. (Cité en page 171.)

[Srikant 2002] Y. N. Srikant et Priti Shankar, editeurs. The compiler design handbook : Optimizations and machine code generation. CRC Press, 2002. (Cité en page 41.)

[Stroustrup 2000] Bjarne Stroustrup. The c++ programming language. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd édition, 2000. (Cité en page 12.)

[Stuckenholz 2005] Alexander Stuckenholz. *Component evolution and versioning state of the art.* SIGSOFT Softw. Eng. Notes, vol. 30, no. 1, page 7, 2005. (Cité en page 14.)

[Sutter 2007] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus et Koen De Bosschere. *Link-time compaction and optimization of ARM executables.* ACM Trans. Embed. Comput. Syst., vol. 6, no. 1, page 5, 2007. (Cité en page 41.)

[Szyperski 2002] Clemens Szyperski. Component software : Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. (Cité en pages 3 et 13.)

[Tassey 2002] Gregory Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing.* Rapport technique, National Institute of Standards and Technology - US Department of Commerce, 2002. (Cité en page 40.)

[The OSGi Alliance 2007] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.1.* http://www.osgi.org/Specifications, 2007. (Cité en pages 12 et 16.)

[Think Team 2009] Think Team. *Think V4 (a.k.a. Nuptse) Programmer's Manual.* http://think.ow2.org/pdf/think-programmer-manual.pdf, 2009. (Cité en pages 145 et 147.)

[ThinkTeam 2010] ThinkTeam. *Think Project.* http://think.objectweb.org, 2010. (Cité en page 29.)

[TinyOS 2010] TinyOS. *TinyOS web site.* http://www.tinyos.net/, 2010. (Cité en page 23.)

[Tiwari 1994] Vivek Tiwari, Sharad Malik et Andrew Wolfe. *Power analysis of embedded software : a first step towards software power minimization.* IEEE Trans. Very Large Scale Integr. Syst., vol. 2, no. 4, pages 437–445, 1994. (Cité en page 41.)

[Tridgell 2000] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, 2000. (Cité en page 60.)

[UltraChip 2010] UltraChip. *UC1601 Passive Matrix LCD Controller-Driver.* http://www.ultrachip.com/upfiles/ADUpload/pro1335548560.pdf, 2010. (Cité en page 153.)

[van Ommering 2000] Rob van Ommering, Frank van der Linden, Jeff Kramer et Jeff Magee. *The Koala Component Model for Consumer Electronics Software.* Computer, vol. 33, no. 3, pages 78–85, 2000. (Cité en pages 19, 21, 23 et 27.)

[van Ommering 2002] Rob van Ommering. *Building product populations with software components.* In ICSE '02 : Proceedings of the 24th International Conference on Software Engineering, pages 255–265, New York, NY, USA, 2002. ACM. (Cité en pages 23 et 27.)

[von Eicken 1992] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein et Klaus Erik Schauser. *Active messages : a mechanism for integrated communication and computation.* In ISCA '92 : Proceedings of the 19th annual international symposium on Computer architecture, pages 256–266, New York, NY, USA, 1992. ACM. (Cité en page 25.)

[von Platen 2006] Carl von Platen et Johan Eker. *Feedback linking : optimizing object code layout for updates.* In LCTES '06 : Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems, pages 2–11, New York, NY, USA, 2006. ACM. (Cité en pages 60, 61 et 64.)

[Voss 2000] Michael J. Voss et Rudolf Eigenmann. *A framework for remote dynamic program optimization.* In DYNAMO '00 : Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, pages 32–40, New York, NY, USA, 2000. ACM. (Cité en pages 38 et 41.)

[Šimunić 2000] Tajana Šimunić, Luca Benini, Giovanni De Micheli et Mat Hans. *Source code optimization and profiling of energy consumption in embedded systems.* In ISSS '00 : Proceedings of the 13th international symposium on System synthesis, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society. (Cité en page 41.)

[WCO 1996] *WCOP'96 : Proceedings of the International Workshop on Component-Oriented Programming*, 1996. (Cité en page 15.)

[Weiser 1993] Mark Weiser. *Some computer science issues in ubiquitous computing.* Commun. ACM, vol. 36, pages 75–84, July 1993. (Cité en page 142.)

[Wigley 2002] A. Wigley, M. Sutton, S. Wheelwright, R. Burbidge et R. Mcloud. Microsoft .net compact framework : Core reference. Microsoft Press, Redmond, WA, USA, 2002. (Cité en page 23.)

[Wilhelm 2008] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat et Per Stenström. *The worst-case execution-time problem—overview of methods and survey of tools.* ACM Trans. Embed. Comput. Syst., vol. 7, no. 3, pages 1–53, 2008. (Cité en page 37.)

[Xie 2006] Qiang Xie, Jinfeng Liu et Pai H. Chou. *Tapper : a lightweight scripting engine for highly constrained wireless sensor nodes.* In IPSN '06 : Proceedings of the 5th international conference on Information processing in sensor networks, pages 342–349, New York, NY, USA, 2006. ACM. (Cité en pages 31 et 53.)

[Zervas 1998] Nikos D. Zervas, Kostas Masselos et C.E. Goutis. *Code Transformations for Embedded Multimedia Applications : Impact on Power and Performance.* In Power-Driven Microarchitecture Workshop In Conjunction With ISCA98, page pp, 1998. (Cité en page 36.)

[Zhang 2009] Zhenyu Zhang, W. K. Chan, T. H. Tse, Heng Lu et Lijun Mei. *Resource prioritization of code optimization techniques for program synthesis of wireless sensor network applications.* J. Syst. Softw., vol. 82, no. 9, pages 1376–1387, 2009. (Cité en page 40.)