

Vers une prise en compte fine de la plate-forme cible dans la construction des systèmes temps réel embarqués critiques par ingénierie des modèles

Olivier Gilles

▶ **To cite this version:**

Olivier Gilles. Vers une prise en compte fine de la plate-forme cible dans la construction des systèmes temps réel embarqués critiques par ingénierie des modèles. Réseaux et télécommunications [cs.NI]. Télécom ParisTech, 2010. Français. NNT : . pastel-00006222

HAL Id: pastel-00006222
https://pastel.hal.science/pastel-00006222

Submitted on 4 Jul 2010

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.

École Nationale Superieure des Télécommunications **Département Informatique & Réseaux**

Département de formation doctorale en informatique **École doctorale de Paris**

Vers une prise en compte fine de la plate-forme cible dans la construction des systèmes temps réel embarqués critiques par ingénierie des modèles

THÈSE

présentée et soutenue publiquement le 1er mars 2010

pour l'obtention du

Doctorat de l'Ecole Nationale Supérieure des Télécommunications

(spécialité informatique et Réseaux)

par

Olivier GILLES

Composition du jury

Président :	Isabelle Demeure	Professeur à Telecom ParisTech
Rapporteurs :	Lionel Seinturier	Professeur à l'université de Lille
	Franck Singhoff	Professeur à l'université de Bretagne Occidentale
Examinateurs :	Nicolas Navet	Chercheur à l'INRIA Nancy Grand Est
	Pierre de Saqui-Sannes	Professeur à l'ISAE
Directeur de thèse :	Jérôme Hugues	Maître de conférence à Telecom ParisTech
Invité :	Jean-Sébastien Cruz, MBDA	

Mis en page avec la classe thloria.

Remerciements

En premier lieu, je tiens à remercier Isabelle Demeure, Lionel Seinturier, Frank Singhoff, Pierre de Saqui-Sannes, Nicolas Navet et Jean-Sébastien Cruz d'avoir bien voulu m'accorder la confiance nécessaire pour être membres de mon jury.

Trois années de travail auxquelles s'ajoutent cinq mois en tant que permanent : le temps passé dans ce qui restera pour moi l'ENST aura été une façon bien singulière et, pour tout dire, assez déraisonnable de conclure mes études.

Rien, en effet, ne me déterminait à suivre cette voie quand, encore lycéen, je formulai mes vœux d'orientation pour les études supérieures avec la certitude qu'il ne s'agissait que d'une formalité administrative. la vie était ailleurs. Il s'est avéré au cours des neuf années qui suivirent qu'elle était également ici...

Le goût des études a toujours été pour moi la conséquence de rencontres. Des hommes et des femmes, mes maîtres, me firent l'honneur de me dispenser leur savoir et leur enthousiasme. A ma grande honte, je n'ai retenu que le nom de quelques-uns d'entre eux. Je me contenterai donc de remercier collectivement les enseignants de l'université Denis Diderot, de l'Institut Universitaire et Technologique d'Orsay et de l'université Pierre et Marie Curie.

Je ne puis, bien sûr, oublier tout ce que je dois à mes camarades — surtout à mes débuts — qui firent de ce travail le contraire d'une ascèse : une aventure joyeuse et vivante. Ils sont nombreux ceux qui ont accompagné ces années, et dont je garde un souvenir ému. Qu'ils sachent que mon amitié fraternelle leur est aquise.

Mes années étudiantes traitées de ces quelques lignes, venons-en à ma thèse. Je tiens évidemment à rendre hommage à Jérôme Hugues, mon directeur de thèse, qui a su répondre à mes doutes avec une discrétion qui lui fait honneur, et sans lequel cette thèse ne serait sûrement jamais arrivée à son terme. Malgré les difficultés matérielles, il a assumé son rôle avec un sérieux aussi rare qu'inattendu. Plus encore que ma gratitude, il y a gagné mon respect. Aussi dérisoires qu'ils soient au regard de ma dette, je lui adresse ces mots "Merci, Jérôme".

Mais mon entreprise de recherche se doublait d'une autre démarche, équilibrant la première. Ma découverte du karatedo m'a permis de garder un équilibre, indispensable, entre pensée et action. Je tiens donc à remercier Jean-Pierre Fischer, mon *sensei*, le club Budokan 13 où je m'entrainais, et le regretté Jean-Luc Halami, qui a combattu jusqu'à son dernier souffle. Entre le dôjô et l'école, je voudrais citer Laurent Pautet, qui m'a recruté, et Gérard Blanchet, mon ainé à l'entrainement et qui me fit l'amitié de relire ma thèse. A tous, *osu* !

D'autres encore marquèrent les derniers jours : philosophes de la machine à café, poètes de restaurant universitaire, érudits éclairés d'un néon tremblottant. Ils furent mes compagnons de bonne et de mauvaise fortune, et partagèrent songes, colères et outrances. Je me suis trouvé avec eux plus de points communs que je ne l'avais cru. Je garderai une pensée pour eux.

Je voudrais, pour finir, remercier mes rapporteurs qui sacrifièrent de leur temps et de leur patience pour apporter les corrections finales à ce document. Merci donc à Lionel Seinturier et à Franck Singhoff pour leur aide précieuse, et merci à tous les autres membres pour leur participation.

Enfin, mes derniers mots seront pour mes parents et mes sœurs. J'ai pu être ingrat parfois, dur souvent. Qu'ils entendent mes excuses, s'ils le veulent bien, et qu'ils me pardonnent, s'ils le peuvent. Nos vies sont trop courtes pour les rancœurs.

À Louis Ferrier et Henri Gilles.

Table des matières

Partie I Problématique **3**

Chapitre 1
Introduction Générale

1.1 Définitions 6
1.1.1 Systèmes Temps-Réel Embarqués 6
1.1.2 Cycle de développement 7
1.2 Un cas d'étude : Generic Avionic Platform 9
1.3 Problématiques liées à l'optimisation 10
1.3.1 Principales contraintes pour l'optimisation 10
1.3.2 Conséquences des contraintes sur le processus d'optimisation 11
1.3.3 Objectifs 11
1.3.4 Contributions 12

Chapitre 2
Problématique et état de l'art

2.1 Optimisation et modélisation 15
2.1.1 Modélisation 15
2.1.2 Génération de code 16
2.1.3 Analyse 16
2.1.4 Plan 17
2.2 Langages de description d'architecture 17
2.2.1 Définition 17

2.2.2 UML/MARTE . . . 18
2.2.3 Fractal/Fractal ADL . . . 19
2.2.4 LwCCM . . . 21
2.2.5 AADL . . . 22
2.2.6 EAST-ADL . . . 24
2.3 Approches pour l'optimisation dirigée par les modèles . . . 25
2.3.1 Éléments caractéristiques . . . 25
2.3.2 OSATE . . . 26
2.3.3 ArcheOpterix . . . 27
2.3.4 SynDex . . . 27
2.3.5 Physical Assembly Mapper . . . 28
2.3.6 Approches partielles . . . 29
2.4 Méthodes de vérification des contraintes . . . 29
2.4.1 Éléments caractéristiques . . . 29
2.4.2 Object Constraint Language . . . 30
2.4.3 Les approches par preuve de théorème . . . 31
2.4.4 Les approches par supervision d'exécution . . . 32
2.5 Positionnement . . . 33
2.5.1 ADL . . . 33
2.5.2 Évaluation des performances et des contraintes . . . 33
2.5.3 Processus d'optimisation . . . 34

Chapitre 3
Optimisation des systèmes Temp-Reél Embarqués

3.1 Relation modèle-performances . . . 37
3.1.1 Modification du comportement temporel . . . 37
3.1.2 Nature du code généré . . . 39
3.2 Opérations sur les modèles . . . 41
3.2.1 Structure du problème . . . 41
3.2.2 Définition du modèle initial . . . 41
3.3 Opération Move . . . 43
3.3.1 Préconditions . . . 45
3.3.2 Règles de nommage . . . 45
3.3.3 Modification des types et interfaces . . . 46
3.3.4 Modifications de la topologie . . . 46
3.3.5 Aspect comportemental . . . 48
3.3.6 Modifications des propriétés non-fonctionnelles . . . 48

3.3.7 Synthèse . . . 48
3.3.8 Postconditions . . . 49
3.4 Opération Merge . . . 49
3.4.1 Préconditions . . . 50
3.4.2 Règles de nommage . . . 50
3.4.3 Modification des types et interfaces . . . 51
3.4.4 Modifications de la topologie . . . 51
3.4.5 Aspect comportemental . . . 54
3.4.6 Propriétés non fonctionnelles . . . 61
3.4.7 Synthèse . . . 63
3.4.8 Postconditions . . . 65
3.5 Autres opérations . . . 66
3.5.1 Opération Split . . . 66
3.6 Conclusion . . . 68

Chapitre 4
Évaluation des modèles

4.1 Optimisation et évaluation . . . 69
4.1.1 Classification du problème . . . 69
4.1.2 Implémentations des techniques d'optimisation . . . 70
4.1.3 Discussion . . . 70
4.2 Algorithmes pour diriger l'optimisation . . . 71
4.2.1 Full-Greedy Heuristic . . . 71
4.2.2 Half-Greedy Heuristic . . . 72
4.3 Contraintes sur les systèmes TRE . . . 74
4.3.1 Ordonnancement . . . 74
4.3.2 Dimensionnement . . . 75
4.3.3 Sûreté . . . 76
4.4 Evaluer une architecture . . . 76
4.4.1 Le calcul des performances avec AADL . . . 77
4.4.2 Niveau d'évaluation . . . 79
4.4.3 Ordonnançabilité . . . 79
4.4.4 Dimensionnement . . . 80
4.4.5 Sûreté . . . 80
4.4.6 Pondération des critères d'évaluation . . . 80
4.4.7 Pipeline d'évaluation . . . 81
4.5 Conclusion . . . 82

Partie II Mise en Œuvre **85**

Chapitre 5
Vérification des contraintes sur les modèles

5.1 Introduction . . . 87
5.2 Structure de REAL . . . 87
5.2.1 Types de bases et ensembles pré-définis . . . 87
5.2.2 Theorem et définition de la portée . . . 88
5.2.3 Construction des ensembles et variables . . . 88
5.2.4 Expressions paramétrées . . . 90
5.2.5 Appel de sous-théorèmes . . . 90
5.2.6 Expression de vérification . . . 91
5.3 Utilisation de REAL . . . 92
5.3.1 Via une annexe AADL . . . 92
5.3.2 Via une bibliothèque de théorèmes . . . 93
5.3.3 Appels de théorèmes . . . 94
5.4 Applications de REAL . . . 94
5.4.1 Vérification de dimensionnement avec REAL . . . 94
5.4.2 Vérification de sécurité avec REAL . . . 96
5.4.3 Assurer la sûreté du mécanisme d'extension de ADDL . . . 99
5.5 Rôle de REAL dans le processus d'optimisation . . . 101

Chapitre 6
Optimisation et évaluation

6.1 Implantation du processus d'optimisation . . . 103
6.1.1 L'architecture Ocarina . . . 103
6.1.2 L'interpréteur REAL . . . 105
6.1.3 Le module Transfo . . . 108
6.2 Construction du modèle enrichi . . . 109
6.2.1 Analyse du WCET et de la pile avec Bound-T . . . 109
6.2.2 Le processus d'enrichissement . . . 111
6.2.3 Déduire la configuration du modèle AADL . . . 112
6.2.4 Annotation du modèle AADL . . . 114
6.2.5 Autres outils . . . 114
6.3 Vérifier la cohérence des modèles avec REAL . . . 114
6.3.1 Les contraintes structurelles . . . 114
6.3.2 Génération de théorèmes . . . 115

6.3.3 Un exemple de contrainte structurelle . . . 116
6.4 Vue d'ensemble du processus d'optimisation . . . 119
6.4.1 Un processus d'optimisation à trois voies . . . 119
6.4.2 Optimisation du modèle . . . 119
6.4.3 Enrichissement du modèle . . . 120
6.4.4 Validation du modèle . . . 120
6.5 Conclusion et avancement . . . 120

Chapitre 7
Cas d'étude et expérimentation

7.1 Generic Avionic Platform . . . 121
7.1.1 La spécification GAP . . . 121
7.1.2 Modélisation avec AADL . . . 124
7.1.3 Le modèle architectural de GAP . . . 126
7.2 Procédure de tests . . . 128
7.2.1 Architecture cible . . . 129
7.2.2 Paramétrage des algorithmes . . . 129
7.2.3 Fonction de coût . . . 132
7.2.4 Fonction d'évaluation de Move . . . 133
7.2.5 Exemple de contrainte sur le modèle . . . 134
7.2.6 Critères d'évaluation du modèle final . . . 134
7.3 Résultats et interprétation . . . 135
7.3.1 Résultats . . . 135
7.3.2 Performances . . . 138
7.4 Synthèse . . . 140

Chapitre 8
Vers un processus de développement basé sur les modèles

8.1 Un processus de développement basé sur l'optimisation des modèles . . . 141
8.2 Contributions . . . 143
8.2.1 Optimisation . . . 143
8.2.2 Langage de description de contraintes . . . 144
8.2.3 Evaluation . . . 144
8.3 Perspectives . . . 145
8.3.1 Améliorations à apporter à l'implantation du processus . . . 145
8.3.2 Traçabilité . . . 146
8.3.3 Vérification comportementale . . . 146

8.3.4 Extension à d'autres langages de modélisation 146

Publications

Annexes

Annexe A
BNF de REAL

Bibliographie **155**

Table des figures

1.1 Processus de développement en spirale . . . 8
1.2 GAP : flots de données . . . 9
1.3 processus d'optimisation à 3 voies . . . 12

2.1 Un composant Fractal . . . 20
2.2 Un composant CCM . . . 22
2.3 Les différentes étapes de l'optimisation . . . 34

3.1 Sous-programmes sérialisables . . . 38
3.2 Sous-programmes sérialisés . . . 38
3.3 Sous-programmes parallélisables . . . 39
3.4 Sous-programmes parallélisés . . . 39
3.5 Équilibrage : exemple initial . . . 40
3.6 Équilibrage : déplacement de sous-programme . . . 40
3.7 Connexion unique entre proc1 (P_{src}) et proc2 (P_{dst}) . . . 44
3.8 Connexion unique entre proc1 (P_{src}) et proc2 (P_{dst}) : Move . . . 44
3.9 Connexion multiple entre proc1 (P_{src}) et proc2 (P_{dst}) . . . 44
3.10 Connexion multiple entre proc1 (P_{src}) et proc2 (P_{dst}) : Move . . . 44
3.11 Connexion entre proc1 (P_{src}) et un processus tiers . . . 45
3.12 Connexion entre proc1 (P_{src}) et un processus tiers : Move . . . 45
3.13 Connexion entre deux threads candidats à un merge . . . 52
3.14 Auto-connexion . . . 52
3.15 Connexion entre un wrapper et un sous-programme unique . . . 52
3.16 Connexion entre un thread à fusionner et un thread non fusionné . . . 53
3.17 Connexion entre un thread à fusionner et un thread non fusionné (résultat) . . . 53
3.18 Connexions multiples . . . 54
3.19 Connexions multiples (résultat) . . . 54
3.20 Utilisation d'un priority_shifter : diagramme de Gantt . . . 62
3.21 Thread contenant 2 call sequences . . . 67
3.22 Split d'un thread contenant 2 call sequences . . . 67
3.23 Thread contenant 2 call sequences et des ports events . . . 67
3.24 Split d'un thread contenant 2 call sequences . . . 67

4.1 Pipeline d'évaluation . . . 82

6.1 Ocarina : organisation générale . . . 103
6.2 Ocarina : organisation du coeur . . . 104
6.3 Inclusion d'un arbre REAL dans un arbre AADL . . . 106

6.4 Dépendances du cœur d'Ocarina pour l'analyse de REAL 106
6.5 Dépendances du cœur d'Ocarina pour l'interprétation de REAL 107
6.6 Positionnement du module Transfo dans Ocarina 108
6.7 Bound-T pipeline 112
6.8 Implementation du processus d'optimisation à 3 voies 119

7.1 Architecture du système GAP : décomposition en senseurs et sous-systèmes, de [69] 123
7.2 Le process Display 127
7.3 Le process Weapons 128
7.4 Le process Navigation 128

Première partie

Problématique

1

Introduction Générale

Alors que les systèmes temps-réel embarqués sont présents de plus en plus fréquemment dans nos vies quotidiennes et gèrent des aspects toujours plus critiques de nos existences — l'apparition et le développement de l'informatique enfouie dans l'automobile ou l'avionique en sont des illustrations exemplaires. Le processus de développement de ces systèmes a toutefois peu évolué durant les 20 dernières années, contrairement au développement logiciel dans des domaines tels que les systèmes d'information ou les systèmes répartis sans temps-réel.
Pourtant, des techniques plus sûres sont apparues (algorithme de contrôle d'accès concurrent PCP, profil de concurrence Ravenscar, patron de sécurité MILS, architecture pour l'avionique modulaire IMA et ARINC653...), et ces techniques tendent à accroître la difficulté du développement en interdisant des techniques courantes de génie logiciel et en demandant une connaissance fine des mécanismes de concurrence et de gestion des ressources. Elles requièrent donc une main-d'œuvre très qualifiée. Plus problématique encore, la taille (en terme de nombre de sous-systèmes) a significativement augmenté. Finalement, les grands projets dans le domaine de l'avionique subissent un accroissement des contraintes sur leurs ressources et leurs délais, car ils intègrent de plus en plus d'acteur, gouvernementaux ou privés.

Ainsi, la conjonction de projets plus complexes, de ressources plus restreintes, de délais plus courts et de méthodes inchangées provoquent une augmentation des retards voire des annulations de projets, avec dans les deux cas des pertes importantes pour toutes les parties impliquées. Ce fut le cas en ce qui concerne les Airbus A380 (18 mois de retard) et A400M (18 mois de retard), le Boeing 787 (3 ans de retard), la version bombardier du F-22 de Lockheed Martin (abandonnée), ainsi que le JSF F-35 de la même firme (dont les surcoûts prévus ont provoqué plusieurs menaces d'annulation de commandes). Si la partie logicielle n'est pas toujours en cause dans ces retards, cela reste le cas dans la plupart des projets (problèmes sur le pilote automatique ou le *Terrain Reference Navigation System* de l'A400M, par exemple).

Un problème fréquemment rencontré dans ce type de systèmes est la difficulté à répondre aux exigences du cahier des charges. Ce sont les exigences non-fonctionnelles (contraintes temporelles, consommation de ressources...) qui posent problème. De part la complexité inhérente à ces systèmes, il est difficile de prévoir a priori leur comportement. Il est au contraire fréquent que l'on doive revenir en arrière dans le développement, non parce qu'une approche est fausse, mais parce qu'elle n'est pas assez efficace. Ces retours en arrière — ou régressions — sont une des raisons les plus importantes des retards de développement pour les composants logiciels.

De façon quelque peu paradoxale, il est envisageable que cette période difficile soit au final une chance pour le domaine, puisqu'elle peut s'avérer un catalyseur de la réflexion sur ses méthodes de développement. Nous sommes en présence d'un marché dynamique dont les demandes ne peuvent

être satisfaites dans les délais exigés. Il existe donc une motivation économique et industrielle suffisante pour que les coûts initiaux d'un changement d'approche soient supportés.

Dans des domaines moins pointus de l'informatique (tels les applications web ou les systèmes d'informations d'entreprises), ces vingt dernières années ont vu l'éclosion puis la généralisation de méthodes de développement partant de postulats assez similaires (complexité croissante des projets, faible engagement des investisseurs, délais extrêmement courts), pour développer des applications fonctionnelles. Ces approches tentaient de réduire les régressions dans le processus de développement Nous pensons qu'une telle approche peut être adaptée dans le cadre des systèmes temps-réel embarqués, à condition d'y ajouter des mécanismes de vérification assurant le maintien de strictes conditions d'exécution.

L'objectif de notre travail de thèse est donc de proposer une approche pour l'articulation d'un processus de développement pour les systèmes temps-réel embarqués limitant au maximum les régressions. Dans ce chapitre, nous présenterons d'abord la définition des différents domaines de validité de cette approche, puis nous définirons rapidement notre cas d'étude, avant de présenter les principaux défis que présente notre objectif.

1.1 Définitions

Dans cette première section, nous présentons les définitions des différentes notions que nous serons amenés à aborder tout au long de notre travail : les systèmes temps-réel embarqués et les cycles de développement.

1.1.1 Systèmes Temps-Réel Embarqués

Les systèmes temps-réel embarqués (TRE) permettent d'interagir avec l'environnement physique en collectant des données en provenance de capteurs, ou en agissant sur celui-ci de façon autonome à l'aide de divers périphériques, dans un contexte de ressources matérielles limitées. Cette rareté des ressources matérielles implique que, en plus de l'aspect fonctionnel (le logiciel doit effectuer les tâches requises), les systèmes TRE doivent prendre en compte le respect des contraintes non-fonctionnelles (délais d'exécution, empreinte mémoire, sûreté...). De tels systèmes sont communément présents dans les domaines tels que l'avionique, le spatial, l'automobile, l'armement ou encore la robotique. Ils ont une importance croissante dans l'économie, car ils permettent d'exercer un contrôle très fin sur les activités industrielles et logistiques.

Les TRE présentent des contraintes spécifiques, qui peuvent être contradictoires ou au contraire se recouper partiellement. Elles sont essentiellement de trois ordres :

Contraintes sur la consommation des ressources

Cette contrainte est essentiellement liée à l'aspect *embarqué* des applications : il s'agit de s'assurer que l'application ne consomme pas plus de ressources logicielles (mémoires, temps processeur...) que l'architecture matérielle n'en fournit effectivement. Si cette contrainte doit être vérifiée sur tout système, elle prend une importance particulière dans un contexte où — pour assurer le déterminisme de l'exécution — l'usage de caches ou d'un système de mémoire virtuelle est limité voire interdit, et où les processeurs culminent à quelques centaines de MHz.

Contraintes spatiales et temporelles

Les systèmes temps-réel agissant sur le monde physique en fonction des données reçues de capteurs, il importe que l'action soit effectuée dans un intervalle de temps qui ne rende pas les données obsolètes. Cet intervalle dépend du domaine considéré — il sera nécessairement différent entre un missile se déplaçant à mach 4 et un cargo naviguant à 20 noeuds — Le strict respect des échéances (ordonnançabilité) doit donc être vérifié.

Contraintes de criticité

De par les tâches qu'ils visent à accomplir, les défaillances dans les systèmes embarqués peuvent avoir des conséquences catastrophiques, que ce soit en termes économiques, écologiques ou civils. De plus, de part la nature autonome d'une grande partie des tâches qu'ils effectuent, il est fréquent qu'un opérateur humain n'ait pas le temps (ou l'occasion) de revenir sur une procédure erronée. Il importe donc que les systèmes TRE offrent des garanties de déterminisme et de tolérance aux pannes.

Optimisation des systèmes TRE

Les critères présentés ci-dessus étant difficiles à concilier, il est fréquent que les premières versions d'un système TRE ne les respectent que partiellement. L'optimisation, qui consiste à modifier le système jusqu'à ce que tous les critères ci-dessus soient vérifiés, est donc une étape nécessaire dans le développement des systèmes TRE.

1.1.2 Cycle de développement

Le processus d'optimisation est une étape parmi d'autres dans le cycle de développement. Il doit donc s'inscrire dans ce cycle, c'est-à-dire qu'une position, une entrée et une sortie doivent lui être affectées. Nous présentons ici quelques éléments de définition sur le cycle de développement, que nous utiliserons par la suite pour localiser dans le temps chaque étape de l'optimisation.

Spécification

Une spécification est le document permettant de définir les contraintes fonctionnelles et non-fonctionnelles. L'étape de conception architecturale permet par la suite d'en déduire l'architecture du système. Dans un milieu industriel, un tel document respecte une forme normalisée. On parlera de *modèle* quand ces spécifications sont interprétables par un logiciel.

Implantation

A partir des spécifications, une implantation du système (nommée *système réel* tout au long de ce mémoire), va être produite, configurée et déployée. Dans le cadre de notre étude nous ne distinguons pas ces trois étapes ; ainsi deux déploiements différents du même système seront traduits par deux systèmes réels différents.

Validation

Une fois le résultat de l'étape d'implantation disponible, il faut tester sa conformité aux spécifications initiales. On notera qu'un échec de cette étape peut avoir deux causes : une erreur

dans la définition des spécifications, ou une de conception. Cette étape est de loin la plus coûteuse du processus de développement — il importe donc de l'automatiser au maximum.

Un processus itératif

De nombreuses méthodes de développement de logiciels existent. Dans le cadre de systèmes embarqués, il est particulièrement important d'en choisir une mettant l'accent sur le contrôle des risques. La méthode de développement en spirale répond à cette contrainte [27]. La figure 1.1 illustre ce type de développement. L'ensemble des étapes est défini sur un plan circulaire, l'idée étant que chaque cycle produise une version du système plus raffinée que le cycle précédent.

Cependant, il est important de noter qu'un échec d'une opération de test est susceptible de provoquer un retour en arrière à l'étape équivalente dans un cycle antérieur — parce qu'elle résulte d'une erreur de conception qui empêche le passage à l'échelle, par exemple. Une erreur dans le développement sera donc d'autant plus coûteuse que l'on s'éloignera des spécifications originelles. C'est pourquoi la détection précoce des erreurs de conception est un enjeu majeur du développement.

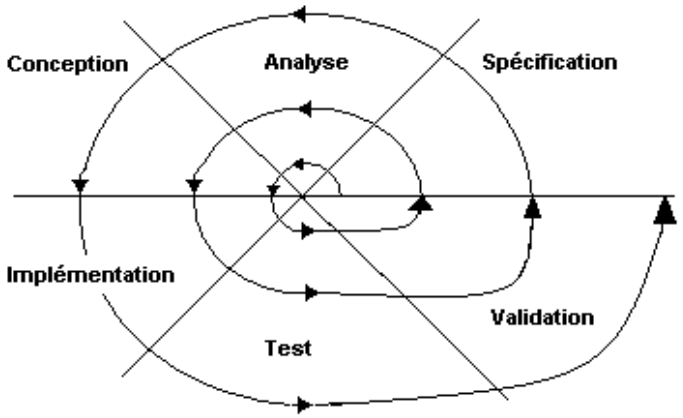


Fig. 1.1 – Processus de développement en spirale

Ce mémoire propose un processus outillé qui permet non seulement d'assurer une détection précoce des erreurs, mais également de minimiser les retours en arrière dans le cycle de développement. On parlera alors de processus de *développement faiblement régressif*.

Un développement dirigé par les modèles

Dans notre travail, nous nous situons dans le cadre d'un développement dirigé par les modèles. Les raisons qui ont mené à ce choix sont les mêmes que celles qui ont motivé les autres pans de l'industrie du logiciel : faciliter la communication entre les équipes de développement, que ce soit de façon instantanée (pour la gestion de projets de taille importante), ou différée (pour la maintenance), et rendre possible toute une gamme de vérifications inenvisageables avec du simple code-source.

1.2 Un cas d'étude : Generic Avionic Platform

Dans le cadre de ce rapport de thèse, nous utilisons un exemple tiré de l'industrie : *the Generic Avionic Platform* pour illustrer comment contruire puis optimiser des systèmes TRE complexes. Cet exemple est une plate-forme définie par le Department of Defense (USA) en 1990 [68], visant à définir les spécifications minimales d'un avion de chasse et d'attaque. Bien que cette plate-forme ne soit plus représentative des systèmes actuels [1], elle reste un exemple réaliste de spécifications industrielles dans un domaine d'application des systèmes TRE.

GAP [2], illustré par le schéma 1.2, définit 16 tâches, périodiques ou sporadiques, aux périodes (ou intervalles minima) différentes, ainsi qu'un grand nombre de connexions. La spécification, en se basant sur des métriques hypothétiques concernant l'implantation, permet d'établir un système ordonnançable (avec RMA)de GAP. En raison de sa complexité, celle-ci suit un découpage fonctionnel.

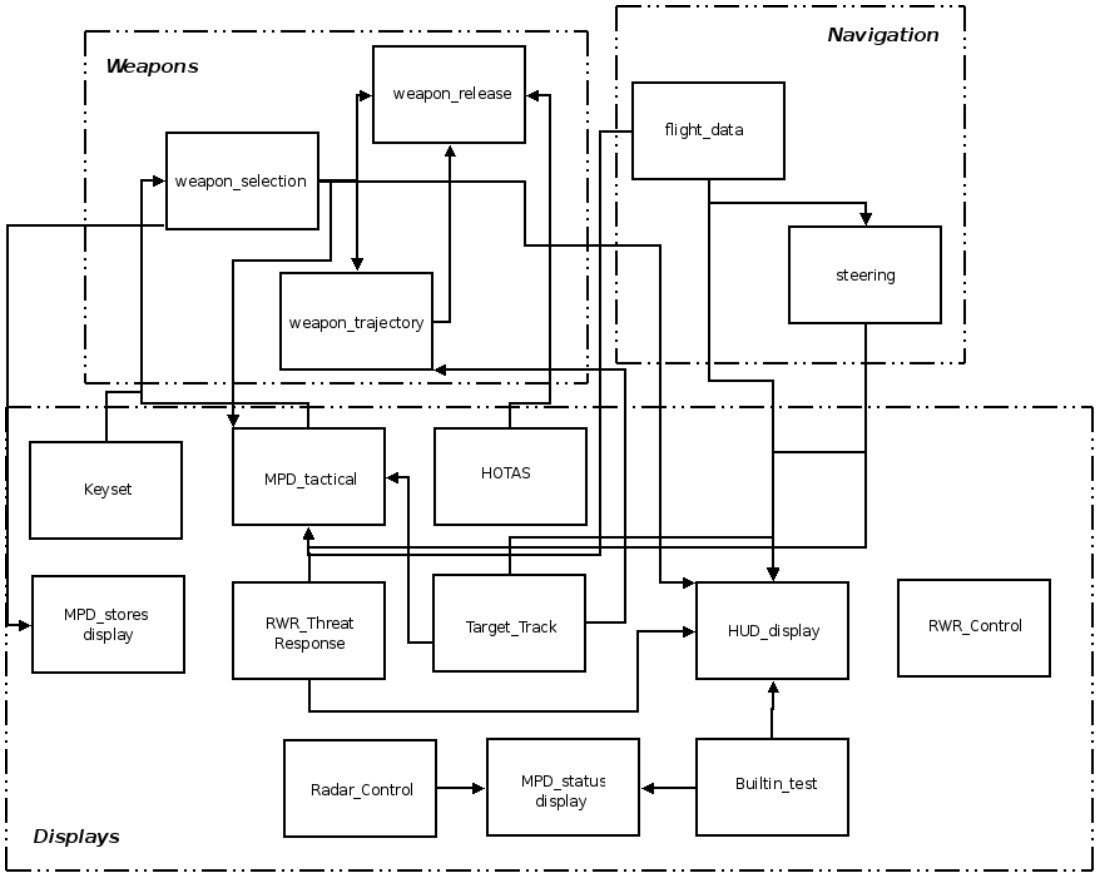


FIG. 1.2 – GAP : flots de données

Nous avons décrit un exemple d'implantation ordonnançable de GAP. Nous proposons d'appliquer notre méthode pour assurer que les propriétés de l'application sont conservées malgré une restriction des ressources matérielles. Pour cela, nous utilisons une étape d'optimisation. Nous verrons également comment nous pouvons assurer la conformité entre le code du système que

[1] Elle ne repose pas sur les concepts de l'avionique modulaire (IMA [66]), dont le standard ARINC 653 est une implantation

[2] Le modèle sera présenté de manière plus complète dans le chapitre 7

nous générons et la spécification fournie par GAP.

1.3 Problématiques liées à l'optimisation

1.3.1 Principales contraintes pour l'optimisation

Mener à bien le processus d'optimisation pour obtenir un résultat répondant aux besoins réels de l'industrie est un problème difficile. Cela induit en effet une phase de développement supplémentaire et donc ajoute en complexité au problème initial. Pour maîtriser cette complexité, nous imposons cinq contraintes principales à notre processus, outre le fait d'accomplir une optimisation effective :

Adéquation de l'optimisation

Pour pouvoir être généralisable, l'optimisation doit se baser sur les contraintes réelles du système, et non sur un critère de performances a priori, car la définition de ces critères dépend intrinsèquement des contraintes et de l'architecture, et il est impossible de définir un critère absolu pour tous les systèmes. Ainsi, dans le domaine de la téléphonie mobile, la consommation d'énergie est un critère important, mais peu pertinent dans le cas de GAP (et de l'avionique en général). La notion d'optimisation dépend donc étroitement du contexte considéré, et ne peut être effectuée qu'avec une connaissance précise de celui-ci.

Flexibilité des critères d'optimisation

Il est possible que l'architecture matérielle ou les contraintes non-fonctionnelles changent au cours de la phase de spécification d'un projet - typiquement, la définition des spécifications et du cahier des charges est un processus itératif qui tend à se poursuivre durant le développement du système, particulièrement en cas de retards. Le coût d'un changement des critères d'optimisation est important, car il représente une somme de travail à la fois important et spécialisé. Il importe donc que le processus d'optimisation puisse changer de critères à moindre coût.

Cohérence implantation - modèle

Les méthodes d'optimisation tendent à réorganiser le système jusqu'à obtenir une combinaison optimale au regard des critères d'évaluation retenus. Cette réorganisation peut avoir un impact sur l'architecture du système final, l'amenant à contredire des spécifications initiales. Ce biais induit une difficulté supplémentaire dans l'analyse des systèmes (puisque les informations des spécifications ne sont plus exploitables), et la maintenance des systèmes. Les systèmes produits par les industriels sont en constante évolution durant une durée de vie qui peut atteindre des dizaines d'années - un exemple typique en est le Mirage 2000-N et sa modernisation en Mirage 2000-D 20 ans après sa mise en service. Il est donc indispensable d'assurer la cohérence des modèles par rapport aux éventuelles modifications apportées par l'optimisation.

Conservation des propriétés des modèles

Dans le cadre d'une approche dirigée par les modèles, il importe de vérifier que le modèle décrit correspond bien aux spécifications initiales, telles que définies en langage naturel dans le cahier des charges. Cet aspect est rendu encore plus important par le fait que l'optimisation peut

être appliquée sur les modèles : si ceux-ci sont transformés, il faut pouvoir offrir une garantie que les propriétés décrites explicitement ou implicitement dans les spécifications seront conservées.

Validation du système réel

Tout processus d'optimisation non-déterministe doit pouvoir offrir un moyen de vérifier la validité des résultats obtenus. Si le processus d'optimisation est basé sur les modèles, il importe de fournir un moyen de vérifier sur le système réel que les performances attendues sont obtenues.

1.3.2 Conséquences des contraintes sur le processus d'optimisation

Au cours des travaux que nous avons mené au cours des trois dernières années, nous avons défini un processus permettant de développer des systèmes TRE à moindre coût, en utilisant des techniques d'optimisation. En fonction des problématiques définies dans la section précédente, nous avons déduit un certain nombre d'indications sur la stratégie à adopter :

Adéquation de l'optimisation Les critères d'évaluation doivent pouvoir être le plus possible adaptés à partir des spécifications du système matériel et, dans une moindre mesure, définis par le développeur système. Un processus d'optimisation doit donc pouvoir assurer l'exploitation du maximum d'information présentes dans les spécifications.

Flexibilité des critères d'optimisation Comme l'indique le point précédent, l'optimisation doit s'adapter au système matériel. Un processus d'optimisation entièrement configurable règle une partie de ce problème. Pour celà, un formalisme générique doit être offert pour spécifier les critères sur lesquels s'effectuent l'optimisation, qui doivent pouvoir être redéfinis par l'architecte.

Cohérence spécification - modèle Un processus d'optimisation qui puisse s'inscrire dans la durée d'un cycle de vie de produit industriel doit assurer que des spécifications utilisables sont produites en même temps que les versions optimisées d'un système.

Conservation des propriétés des modèles Une méthode générique et réutilisable pour spécifier des contraintes doit être offerte à l'architecte, de manière à minimiser l'impact d'un changement de spécification. Les différentes versions du modèle initial doivent toutes vérifier les propriétés définies par l'architecte ou déduites de la structure du modèle original. En particulier, une contrainte définie sur un système donné doit pouvoir être vérifiée sur un autre système sans réécriture.

Validation du système réel La validation doit être effectuée non sur les valeurs estimées des différentes caractéristiques non-fonctionnelles (puisqu'elles ont déjà été utilisées pour l'optimisation), mais sur les valeurs réelles, calculées sur le système réel à partir d'outils indépendants.

1.3.3 Objectifs

Il ressort des observations précédentes que le processus d'optimisation doit être centré sur les spécifications. Pour que ces spécifications soient exploitables de façon automatisée, elles doivent être décrites sous la forme d'un modèle. Le langage de modélisation utilisé sera alors nommé *language-pivot* du processus d'optimisation. Exécuter l'optimisation sur ce modèle doit permettre

d'obtenir un nouveau système (ici un modèle de système) sans perte d'information. Nous pouvons assurer la cohérence du système réel optimisé avec le modèle optimisé en générant le code final (ainsi que la configuration et le déploiement) à partir du modèle. Finalement, un langage d'expression de contrainte spécifique à notre langage-pivot permet de diriger à la fois le processus d'optimisation et celui d'évaluation.

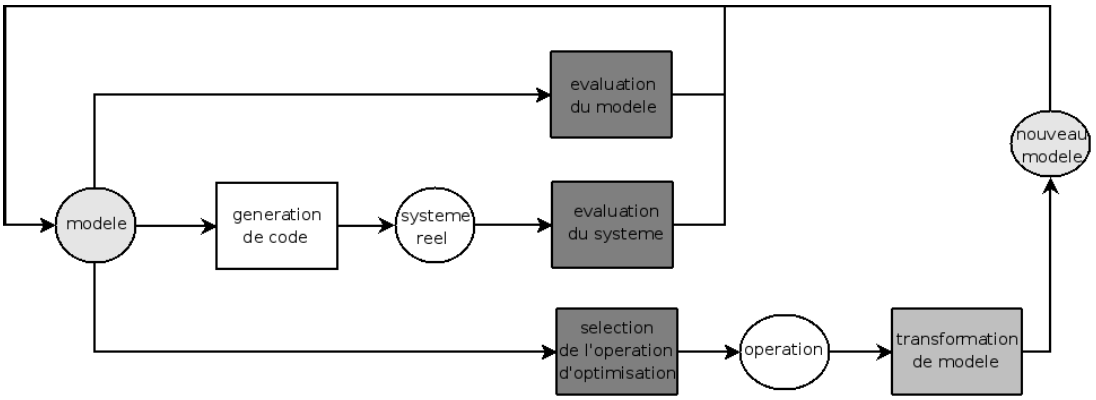


FIG. 1.3 – processus d'optimisation à 3 voies

Le schéma 1.3 illustre les différentes étapes du processus que nous proposons. Nous voyons que dans ce processus, trois actions sont possibles à chaque itération : optimisation, évaluation au niveau du modèle ou évaluation au niveau du système réel (les binaires).

Notre objectif est de factoriser la partie relevant de l'évaluation (au sens général) dans ces trois parties (rectangles gris foncés dans le schéma). Le chapitre 4 décrit en détail cette problématique. Pour parvenir à ce résultat, nous utilisons un langage-pivot, AADL (standardisé par le SAE), ainsi qu'un langage de description de contrainte dédié à ce langage que nous avons défini, REAL. Ces éléments sont motivés dans le chapitre 2. REAL est décrit précisément dans le chapitre 5 de la seconde partie de ce mémoire. Les différentes techniques utilisées pour l'optimisation (en gris intermédiaire dans le schéma) sont présentées dans le chapitre 3. Finalement, nous présentons dans le chapitre 6 la façon dont nous implantons les différents points présentés dans les chapitres précédents, et comment nous les intégrons dans un processus d'optimisation complet. Le chapitre 7 illustre l'application de cette approche à notre cas d'étude. Nous concluons dans le chapitre 8, et synthétisons les avantages de notre approche, avant de présenter les améliorations possibles et quelques approches complémentaires.

1.3.4 Contributions

Nous verrons dans ce mémoire comment nous pouvons mettre en place ces différentes phases ainsi que leur mise en œuvre. Plus particulièrement, nous apportons les contributions suivantes à l'optimisation des systèmes TRE :

- nous présentons des méthodes d'optimisation sur les modèles et étudions leur impact sur les performances du système réel (chapitre 3) ;
- nous montrons comment un langage-pivot peut servir de fil directeur au processus d'optimisation (chapitre 4) ;
- nous présentons diverses solutions pour l'évaluation des systèmes TRE (chapitre 4) ;
- nous proposons REAL, un langage de description de contrainte spécifique à notre langage-pivot, et montrons les avantages qu'il y a à un telle approche (chapitre 5) ;

- nous montrons comment un langage de description de contraintes sur les modèles permet de diriger à la fois les algorithmes d'optimisation et l'évaluation du système et assure une flexibilité importante à l'optimisation (chapitre 6) ;
- nous avons déduit de ce processus une implantation en Ada intégrée dans l'outil Ocarina (chapitre 6) ;
- nous proposons un processus de configuration des outils d'analyse à partir de la description architecturale, et donnons un exemple d'implantation l'appliquant à l'analyse du pire temps d'exécution (WCET) (chapitre 6) ;
- nous proposons une approche permettant de vérifier la cohérence structurelle entre les différentes versions du modèle avec REAL (chapitre 6) ;

Nous validons notre approche dans le chapitre 7, en l'appliquant à une version multi-processeur avec communications asynchrones de la spécification GAP, et nous montrons comment nous pouvons améliorer les performances de ce dernier grâce à notre processus d'optimisation. Grâce à ce dernier, nous arrivons à un gain en empreinte mémoire de 30% à 40%, tout en conservant à la fois l'ordonnançabilité et les contraintes initialement définies sur le modèle, et en prouvant cette conservation.

Nous concluons ensuite sur notre travail dans le chapitre 8, en présentant précisément nos différentes contributions et leur place dans un processus d'optimisation, puis différentes améliorations qui peuvent lui être fait.

2

Problématique et état de l'art

La définition, la validation, la génération et le déploiement de systèmes TRE nécessitent la connaissance d'un certain nombre d'informations importantes sur ces systèmes :

- informations d'ordre topologique, indispensables pour le déploiement et l'analyse ;
- informations relevant du dimensionnement, également nécessaires au déploiement et à l'analyse, mais également à la génération de code ;
- informations comportementales, indispensables à l'analyse et à la génération ;
- informations sur les contraintes à respecter, indispensables à la validation.

Le support de ces informations, cependant, n'est pas suffisant pour justifier le choix d'un paradigme particulier : encore faut-il que cette information soit simplement (c'est-à-dire si possible automatiquement) accessible et associable à l'élément correspondant du système, sans quoi aucun procédé de développement automatisé ne peut être envisagé.

2.1 Optimisation et modélisation

2.1.1 Modélisation

Nous avons vu dans le chapitre 1 que cinq contraintes pèsent sur l'optimisation des systèmes TRE si l'objectif est de mettre en place un processus généralisable : cette optimisation doit être effectuée selon des critères issus de l'analyse du problème réel, et non décidée a priori ; elle doit pouvoir être contrôlée par l'architecte, et elle doit offrir une méthode permettant d'assurer la compatibilité avec les spécifications initiales ; elle doit assurer la cohérence entre les différents modèles ; elle doit offrir une technique pour valider les performances effectives des systèmes réels.

Pour réaliser ce processus d'optimisation, et pour assurer la conformité aux exigences que nous venons de définir, nous ne pouvons que nous appuyer sur un modèle. En effet, à chaque étape du processus, il faut pouvoir analyser les contraintes issues de l'architecture logicielle et matérielle pour proposer une procédure d'optimisation adaptée. Cette analyse demande une représentation symbolique de la solution considérée, c'est-à-dire un modèle, au sens large.

Beaucoup de formalismes utilisés pour représenter une solution logicielle, à commencer par les langages de programmation, adoptent une vue fonctionnelle de celle-ci. Or, une telle vue, dont le destinataire final est un processeur ou un prouveur, n'est pas appropriée pour définir le dimensionnement ou les interactions présents dans une application. Certaines vues partielles offriront au contraire une représentation des aspects topologiques (par exemple les Réseaux de Petri qui associent une représentation fonctionnelle et topologique). Dans les deux cas, cependant, la possibilité de définir des composants structurés, de les déployer sur une architecture matérielle et d'en examiner les propriétés n'est pas offerte.

Les langages de description architecturaux (ADLs) proposent une structure de modélisation hiérarchique, assurant que toute information peut être automatiquement retrouvée. Ils sont donc tout indiqués pour servir de langage-pivot à un processus de développement et d'optimisation automatisé.

2.1.2 Génération de code

Si la modélisation permet de conserver et de retrouver les informations concernant l'architecture, elle implique également un risque supplémentaire : celui d'une divergence entre le modèle et le système réel. Ce risque, déjà présent dans la phase initiale d'implantation, devient d'autant plus important qu'on s'éloigne, du développement du modèle initial. C'est donc un aspect particulièrement critique dans le cadre d'une maintenance devant s'étaler sur plusieurs années, voire plusieurs décénies — un ordre de grandeur tout à fait courant dans le domaine de l'avionique.

La génération de code permet de palier ce problème. Il s'agit de générer le code automatiquement à partir du modèle, ce qui ramène la maintenance à un problème de modélisation. Une telle approche ne peut évidement être mise en place que si il existe une correspondance documentée entre chaque élément du langage de modélisation et des motifs de code. L'existence d'une telle correspondance est donc un critère primordial pour la sélection d'un langage de modélisation.

Il est a noter que certaines optimisations peuvent être effectuées au niveau du code généré, sans connaissance du modèle. De telles optimisations relèvent d'une approche proche de celle des compilateurs. Dans notre thèse, nous avons exclu cette problématique, la jugeant largement étudiée — les auteurs de [47] fournissent un exemple d'une telle étude dans le contexte des langages synchrones. Des améliorations peuvent donc être apportées à ce niveau, sans remettre en cause notre approche.

2.1.3 Analyse

Toute procédure d'optimisation globale (appliquée aux modèles ou non) doit être munie d'une fonction d'évaluation, de manière à pouvoir contrôler l'optimisation en sélectionnant les solutions optimales. Nous distinguons dans ces fonctions d'évaluation deux niveaux.

Les fonctions d'évaluation de bas niveau permettent de récupérer des informations sur le système réel. Il s'agit typiquement de connaitre le WCET ou l'occupation mémoire d'un sous-programme, la mémoire disponible pour un processus, etc. De telles valeurs peuvent dépendre de l'implantation (mais dépendent du modèle si on pratique la génération de code), du déploiement ou uniquement du profil du code-métier initial. Dans les deux derniers cas, ces informations peuvent être extraites d'un modèle, en particulier si celui-ci est architectural (puisque le typage d'un composant permet de retrouver les informations qui lui sont associées).

Dans le cas d'informations dépendant du modèle ou de l'implémentation (mais pas du déploiement), retrouver cette information demande une analyse sur les binaires du systèmes réel. Cette analyse peut être très simple (par exemple, mesurer la taille de la partie `data` de la mémoire d'un sous-programme ne demande que de mesurer une section contigüe dans le fichier objet généré). D'autres mesures, au contraire, sont intrinsèquement difficiles (par exemple, l'analyse précise du WCET implique non seulement de prévoir les effets d'un éventuel cache, de branchements conditionnels, mais également le coût des différentes primitives systèmes). Dans les deux cas, l'analyse des binaires est indispensable. Comme cet aspect échappe à la portée de notre thèse, et, surtout, est déjà lrgement traité [90], nous nous sommes reposés sur des outils pour mener cette action. Au cours de notre thèse, nous montrons comment, une fois cette information obtenue,

nous pouvons la réinjecter dans le modèle. La possibilité d'annoter le modèle est donc également une caractéristique indispensable pour le langage de modélisation que nous sélectionnerons.

Les fonctions d'évaluation de haut niveau permettent d'associer les informations obtenues par les fonctions de bas niveau et d'en déduire une valeur qui sera associée au modèle en cours d'évaluation. Les fonctions d'évaluation doivent donc pouvoir accéder aux information contenues dans le modèle. En plus de la possibilité d'annoter un modèle, il est donc indispensable de pouvoir simplement retrouver une information par rapport à un élément spécifique du modèle, comme le permettent les ADLs. Nous verrons également comment ces derniers permettent d'effectuer des comparaisons ou des calculs sur ces valeurs.

2.1.4 Plan

Dans le présent chapitre, nous proposons donc une étude des principaux ADLs, en essayant de capturer les caractéritiques qui seront utiles dans un processus d'optimisation. Nous passons ensuite en revue quelques méthodes d'optimisation, basées sur les modèles ou non, et exposons les intérêts et faiblesses de ces approches dans le cadre de notre étude. Par la suite, nous présentons les principales techniques d'évaluation de haut niveau — ou méthodes de descriptions de contraintes — actuellement disponibles, leur atouts et leurs limites.
Finalement, nous définirons une méthode originale tirant parti à la fois des informations présentes au niveau modèle, et celles présentes au niveau du systeme reel

2.2 Langages de description d'architecture

Un langage de description architectural (ADL) permet de spécifier à la fois la topologie, le dimensionnement et la hiérarchie des composants logiciels d'un système. Nous traitons dans cette section des critères de classification des ADL, puis présentons un certain nombre de langages significatifs.

2.2.1 Définition

Nous présentons ici les éléments caractéristiques d'un ADL, en nous appuyant sur le travaux de [73], puis nous exposons les caractéristiques qui nous intéressent dans le cadre d'un processus d'optimisation dirigé par les modèles.

Éléments caractéristiques des ADLs

Les composants sont des entités (généralement logicielles), actives ou passives. Il s'agit par exemple d'une unité de stockage de donnée ou de calcul. Par définition, les ADLs autorisent l'usage de composants hiérarchiques, qui contiennent d'autres composants. Dans beaucoup de cas, tout composant est potentiellement hiérarchique. La notion d'héritage, venue de l'orienté objet, est généralement associée aux composants, permettant de dériver un composant d'un autre.

Les connecteurs Il s'agit de la partie du modèle qui décrit la topologie du système - même si dans le cas de descriptions incluant le matériel, certains composants puissent également participer à la description topologique. Les connecteurs peuvent représenter un grand nombre d'opérations, depuis l'affectation dans une variable locale jusqu'à un appel de procédure à distance. Ils offrent

(du côté de l'émetteur) et requièrent (du côté du recepteur) des interfaces pour les différentes communications.

La hiérarchie Il s'agit de la relation hiérarchique entre les différents composants. Dans le cadre de cette étude, nous incluons les informations relatives au déploiement de l'architecture logicielle dans la notion de hiérarchie. Cette information est indispensable non seulement pour la génération et le déploiement de l'application, mais permet de structurer le modèle, et donc de rendre possible la désignation d'éléments spécifiques de celui-ci, par exemple dans un but d'analyse.

Caractéristiques retenues

Nous présentons ici les différentes caractéristiques que nous avons retenues dans le but de diriger un processus d'optimisation basé sur les modèles. Ces caractéristiques, complémentaires de celles propres aux ADLs, permettent de faciliter à la fois l'adoption du processus et la réalisation des différentes opérations de ce dernier : évaluation, transformation/optimisation et analyses spécifiques.

Standardisation La standardisation d'un langage permet de simplifier les interactions entre les développeurs et de faire adopter de manière pérenne l'ADL dans les milieux industriels. Il est très difficile de garantir l'interopérabilité des outils développés par des tiers si aucune standardisation n'existe. Il s'agit donc d'un critère important dans le cadre d'une approche outillée. En particulier, cela permet de disposer de plus d'outils d'analyse, et donc de raffiner l'évaluation.

Homogénéité de la description Certains ADLs proposent plusieurs formalismes complémentaires pour décrire l'application. Si une telle approche permet de définir précisément tous les aspects de l'application, elle fait apparaitre un risque de divergence entre les différents modèles, en particulier dans un processus de raffinement. Une telle approche est donc préjudiciable dans notre cas.

Possibilité d'annotation En plus des informations nécessaires aux spécifications des composants, la plupart des langages offre la possibilité d'ajouter des informations supplémentaires sur divers éléments du modèle. Cette possibilité est indispensable pour ajouter les informations issues du processus d'optimisation, ou tout simplement pour décrire précisément l'application.

Capacité d'analyse Finalement, les capacités d'analyse liées à la structure du langage détermineront le processus d'évaluation du système. Elles sont donc cruciales dans notre cas.

2.2.2 UML/MARTE

MARTE [41] (*UML profile for Modeling and analysis of Real-Time and Embedded systems*) est, comme son nom l'indique, le profil officiel du langage de modélisation UML pour le temps-réel et l'embarqué. UML (*Unified Modeling Language*) a pour but de fournir un langage de modélisation standard pour un vaste éventail de systèmes, sans se limiter à la modélisation d'applications logicielles. Sa vocation est d'unifier les approches qui l'ont précédé (OMT, Jacobson Use Case, etc.), et donc de couvrir toutes les phases du développement logiciel, depuis la rédaction du cahier des charges jusqu'à l'implantation et le déploiement.

Les composants Le composant UML se nomme *classe*. Une classe déclare ses *interfaces*. Ces dernières permettront de déclencher l'exécution de méthodes par un *objet* (instance d'une classe), de la même manière que dans l'orienté objet. Une méthode peut changer l'état de l'objet, fournir des informations à son déclencheur. Une interface peut être *fournie* (présente dans une classe) ou *requise* (demandée par une classe). Il est donc possible de poser des contraintes sur les connexions inter-composants.

Un port est un élément d'une classe, permettant d'encapsuler des interfaces fournies ou requises. Les connexions entre les objets peuvent donc se faire par leurs ports, s'ils fournissent et requièrent les même interfaces.

Les connecteurs Le connecteur va permettre de spécifier une chemin possible de communication entre deux classes.

La hiérarchie La hiérarchie s'effectue soit au travers de connecteurs, soit au travers de différents diagrammes : le diagramme de déploiement (qui décrit le matériel sur lequel s'exécute l'application, et le déploiement de celle-ci), de communication (qui décrit les flots de données entre les composants) ou de package (qui décrit la hiérarchie et les dépendances entre les éléments).

Standardisation UML/MARTE est standardisé par l'OMG [3](*Object Management Group*), un consortium international existant depuis 1989 et dont la pérennité et l'influence sont une garantie pour l'architecte.

Homogénéité de la description UML/MARTE propose un nombre important de *vues* différentes, couvrant ainsi tous les aspects du développement, mais posant des problèmes de cohérence.

Possibilité d'annotation Dans le diagramme de classe, il est possible de représenter les parties internes d'une classe grâce a un diagramme spécifique, dit de "structures composites". Ces parties sont les attributs des classes, et peuvent servir à ajouter une propriété à celles-ci. Ce mécanisme s'applique également aux diagrammes de séquences et d'activité.

Capacité d'analyse L'utilisation du profil MARTE permet de d'effectuer des analyses temporelles [71]. Des possibilités d'analyse du dimensionnement sont également possibles, mais exigent de naviguer entre les vues. Peu d'outils existent à ce jour pour effectuer ces analyses.

2.2.3 Fractal/Fractal ADL

Le modèle de composant Fractal, dont la première version a été distribuée en 2002, est un modèle de composant dont le but est d'implémenter, de déployer et de gérer (i.e. surveiller, contrôler et reconfigurer dynamiquement) des systèmes complexes. On notera que cette capacité est particulièrement appropriée dans le cadre de systèmes temps-réel "souples" (dont on peut manquer une partie des échéances), mais n'est pas adaptée aux systèmes temps-réel durs et/ou critiques. Cela, pour autant, ne disqualifie pas le langage.

Fractal [31] est un modèle de composant qui peut être implanté dans différents langages de programmation. Sa portée s'étend sur tout le processus de développement, depuis la conception architecturale jusqu'à l'implémentation, le déploiement et la configuration. Il a été utilisé pour construire différents types de systèmes : noyaux de systèmes d'exploitation, serveurs d'application, applications multimédia, etc.

En plus d'une notion de hiérarchie commune aux ADLs — nommée *récursivité* — le modèle de composant Fractal introduit la notion de composant doté d'un ensemble d'interfaces de contrôle (*réflexivité*). En d'autres termes, l'exécution et la structure interne des composants Fractal peuvent être explicitées et contrôlées par des interfaces bien définies.

Composants Un composant Fractal est une entité d'exécution encapsulée et propose une ou plusieurs interfaces. Une interface est un point d'accès qui implante un *type d'interface* spécifiant les opérations supportées. Parmi les composants, on distingue :

- des composants composites (contenant des sous-composants), dans le but d'avoir une vision uniforme des applications à différents niveaux d'abstraction et assurant la possibilité de récursivité ;
- des composants partagés (sous-composants appartenant à différents composants), dans le but de modéliser les ressources partagées.

Tous ces composants sont pourvus d'une capacité d'introspection, pour surveiller et contrôler l'exécution du système, et d'une capacité de reconfiguration dynamique du système.

Un composant Fractal, est doté d'une membrane, qui supporte les interfaces pour l'introspection et la reconfiguration de ses propres attributs, et d'un contenu, qui correspond à un ensemble fini d'autres instances de composants (des sous-composants). La membrane offre deux types d'interfaces : les interfaces internes, qui sont fournies aux sous-composants, et les interfaces externes, qui sont fournies aux composants externes. La figure 2.1 illustre un composant Fractal.

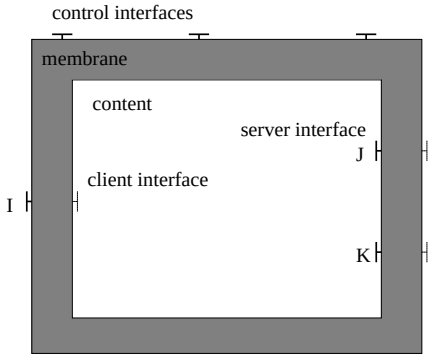


Fig. 2.1 – Un composant Fractal

Connecteurs Le modèle Fractal fournit un mécanisme unique pour définir les connexions entre les différents composants d'une application : la liaison (*binding*). Un composant Fractal peut invoquer une primitive sur une interface serveur d'un autre composant avec une de ses interfaces clients si ces deux interfaces sont liées. Ces liaisons sont construites à partir d'un ensemble de primitives de liaisons et de composants de liaisons : souches, squelettes, adaptateurs, etc. Une liaison est un composant Fractal à part entière, et est donc dotée de capacités d'introspection et de reconfiguration (réflexivité).

Hiérarchie Comme vu ci-dessus et comme son nom l'indique, le modèle Fractal intègre la notion de sous-composant, et donc de hiérarchie.

Standardisation Fractal est supervisé par ObjectWeb, un consortium pour le logiciel libre créé en 1999 par Bull, France Telecom et l'INRIA. On remarquera au passage que tous les acteurs

majeurs sont français, ce qui pose question sur sa portée internationale. Si Fractal est utilisé par de nombreux industriels, il n'existe cependant pas de standard le définissant.

Homogénéité de la description Fractal ADL offre la possibilité de définir le modèle de composant en un unique document. La description d'une composant est donc homogène.

Possibilité d'annotation Le méta-modèle de Fractal ne prévoit pas d'annotations d'un composant.

Capacité d'analyse Fractal ADL se voulant un langage générale de description d'architecture, il ne permet pas de disposer d'informations suffisamment précises pour procéder à des analyses sur le système. Il se focalise en effet sur les descriptions hiérarchiques entre composants, ainsi que sur leurs interfaces. C'est pourquoi il existe des démarches comme [21] qui se basent sur Fractal ADL pour faire de l'analyse comportementale via des annotations supplémentaires.

2.2.4 LwCCM

CCM [7] (*CORBA Component Model*) définit un modèle de composant qui peut être utilisé en conjonction avec D&C [10] (*Deployment and Configuration of Component-Based Distributed application*) pour spécifier un déploiement sur cible. Lightweight CCM (LwCCM) est un profil CCM pour les systèmes temps-réel et embarqués [8].

L'approche CCM préconise de produire des éléments de modélisation et d'exécution réutilisables, car un grand nombre de modules CORBA requièrt au final les mêmes services. Plus généralement, elle vise à fournir un environnement pour la conception, le développement, le déploiement et l'exécution des modules CORBA. Pour arriver à cet objectif, le standard CCM met en avant :

- la stricte séparation contraintes fonctionnelles/non-fonctionnelles ;
- la composition des composants ;
- le support du cycle de vie des composants.

Les composants Un composant LwCCM a un type et des ports de communication. Il fournit et requièrt des interfaces, et peut envoyer ou recevoir directement des données (autorisant donc des motifs de communication par envoi de message ou par RPC). Des *attributs* (variables) sont également accessibles des composants externes, et utilisés pour la configuration des composants.

Les connecteurs les *facettes* permettent d'offrir des interfaces, tandis que les *réceptacles* permettent de les requérir. Ce type de connectivité implique un typage. Les envois et réception de messages asynchrones passent respectivement par des *sources* et des *puits*. La figure 2.2 illustre un composant CCM muni de chacun de ces types de connexions, en entrée et en sortie.

La hiérarchie Un composant CCM peut hériter d'autres composants, ou être l'hôte d'une instance de composant. La notion de hiérarchie existe donc sous deux formes distinctes.

Standardisation LwCCM est standardisé par l'OMG.

Homogénéité de la description Comme UML/MARTE, LwCCM repose sur de multiples vues du même modèle.

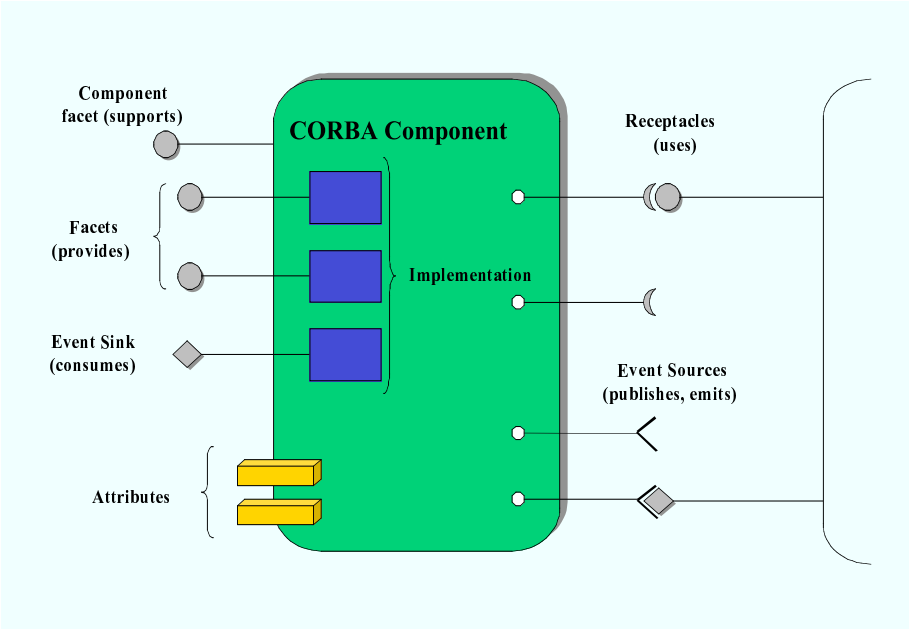


FIG. 2.2 – Un composant CCM

Possibilité d'annotation S'il n'est pas possible d'ajouter des informations exploitables par un processus de développement aux composants CCM, des variantes du standard (comme MyCCM [29] ou RT-CCM [72]) permettent d'effectuer des annotations sur les composants.

Capacité d'analyse La configuration s'effectue via trois catégories de modèles :

- le modèle de composants, qui décrit les composants, leur implémentation et les exigences de configuration et de déploiement ;
- le modèle-cible décrit la plate-forme d'exécution (matérielle et logicielle) sur laquelle les composants seront déployés ;
- le modèle d'exécution, qui spécifie le déploiement.

Ainsi, il est possible de définir toute l'architecture logicielle et matérielle de l'application - bien que cette description soit extrêmement éloignée des contraintes de ressources, ce qui la rend relativement inexploitable. On notera cependant que des contraintes peuvent être définies quant aux services offerts par les sous-composants.

2.2.5 AADL

Le langage AADL (pour *Architecture Analysis and Design Language*) est un ADL basé sur la notation textuelle de MetaH [67]. La dernière version 2 du langage a été produite en 2009 [12], remplaçant la version antérieure datant de 2004 [9]. Ce langage est conçu pour la spécification des systèmes embarqués et temps-réel, et plus particulièrement pour le domaine de l'avionique. AADL a été développé en gardant en tête les problèmes d'analyse de système : il facilite de fait l'interopérabilité d'outils.

Composants Un modèle AADL repose sur une hiérarchie de composants typés et interconnectés. Chaque composant dispose d'interfaces liées à une ou plusieurs implantations. Il y a trois familles de composants : logiciels, matériels ou hybrides. Les types de composants logiciels reflètent

les grands concepts de l'ingénierie système : threads, processus, sous-programmes, données, etc. Ils permettent ainsi de garder une vue proche des questions d'implémentation et de déploiement, critère indispensable pour modéliser des systèmes à fortes contraintes de ressources. Les composants matériels permettent de modéliser l'architecture matérielle : bus, données, processeurs et périphériques. Finalement, les composants hybrides (*systems*) peuvent contenir d'autres composants. Dans la seconde version du langage, des composants abstraits ou virtuels autorisent un développement itératif par raffinement des composants. Un composant peut être mis en relation avec d'autres composants par ses connecteurs, mais également avec une relation hiérarchique.

Connecteurs Une interface de composants peut contenir différents éléments (nommés *features*). Une feature peut être :

- un port (ou interface de communication). Un port peut être en entrée (in), en sortie (out), ou bidirectionnel (inout). Il peut transmettre un événement, une donnée, ou le deux à la fois ;
- des paramètres (qui se comportent comme des ports de données) ;
- des groupes de ports ;
- des informations indiquant si un composant requiert ou fournit des accès spécifiques (bus par exemple).

Les features de deux composants sont reliées entre elles par des *connexions*.

Hiérarchie AADL permet de hiérarchiser les éléments, et impose des règles strictes pour cette hiérarchie. Un type de composant peut ainsi contenir certains autres types de composants, mais pas tous (par exemple, impossible de mettre un processus dans une donnée).

Standardisation La standardisation d'AADL est gérée par la SAE International (Society of Automotive Engineers). Cette organisation a pour but de développer des standards depuis 1905. AADL est donc standardisé au niveau international.

Homogénéité de la description AADL n'impose qu'un unique modèle. Cette vue réellement centrée sur l'architecture (comparativement aux autres ADLs) constitue l'un de ses meilleurs atouts.

Possibilité d'annotation Il est possible d'ajouter des propriétés typées à la plupart des éléments de AADL, dont les composants, les features et les connexions.

Capacité d'analyse AADL est un langage qui se prête particulièrement bien à l'analyse où il est possible de disposer d'informations relatives :

- aux communications ;
- au déploiement ;
- aux temps ;
- aux threads ;
- à la mémoire ;
- à l'ordonnancement ;
- aux appels de programme ;
- à la gestion des erreurs.

La modélisation de ces propriétés est intégrée dans le standard AADL.

2.2.6 EAST-ADL

EAST-ADL [17] est un langage architectural spécialisé pour l'automobile. Comme AADL, il couvre aussi bien les aspects logiciels que matériels. Il permet d'exprimer nativement l'aspect comportmental du système, et de lui associer des contraintes. Ce langage définit le concept de *niveaux d'abstraction*, correspondants à des étapes couramment définies par les processus de développement (tests physique, implémentation, modélisation...). Les contraintes peuvent être suivies et vérifiées à travers les différents niveaux d'abstractions. ces contraintes doivent cependant porter sur des aspects fonctionnels ou causaux du système : le dimensionnement, en particulier, échappe à sa portée. Une attention particulière a été apportée au support des outils existants. En ce sens, EAST est autant un processus de développement qu'un langage architectural.

Les composants Cinq catégories de composants existent dans EAST-ADL : *structure*, *behaviour*, *requirements*, *validation* et *variable handling*. Seule la première catégorie relève de l'architecture. La nature d'un composant varie selon le niveau d'abstraction considéré. Au premier niveau (*acausal*), il s'agit essentiellement d'un équation algébrique ou différentielle, potentiellement associée à un automate à états. Le second niveau (*causal*) ajoute la notion de type passant sur les connections. Le troisième niveau (*discretized*) offre une version discrète du niveau précédent, sous la forme d'une représentation algorithmique. Le dernier niveau, *simulation*, est comme son nom l'indique l'implémentation d'un simulateur, composé d'un ordonnanceur et d'un solveur. Dans la pratique, tous les niveaux d'abstractions ne sont pas nécessairement visibles par l'architecte. Par exemple, les auteurs de [84] proposent de se limiter à la définition du modèle *causal* en utilisant des diagrammes SysML. On notera que la vue proposée à l'architecte est implicitement fonctionnelle, c'est-à-dire que le découpage de l'architecture est supposé suivre les fonctionnalités offertes par l'application.

Les connecteurs Des règles de connexions existent entre les différents types de composants, comme en AADL. Le niveau de détail des communications est relativement précis, puisqu'il descend au niveau des frames — la domination du standard CAN dans le domaine automobile permettant d'offir des motifs de communications plus précis que dans des langages à vocation plus générale.

La hiérarchie EAST-ADL supporte la notion de *variabilité*, que l'on peut associer à de l'héritage. Elle permet de conserver une partie des preuves obtenues sur les composants dont on dérive un composant. Des règles d'associations existent également entre les différents types de composants.

Standardisation EAST-ADL étant spécifiquement conçu pour les systèmes automobiles, il est compatible avec le standard de facto du domaine, AUTOSAR [19], au sens où il utilise un sous-ensemble des concepts de ce dernier. Concernant les standards moins spécifiques à un domaine, EAST-ADL s'inspire de SysML (en particulier de ses apports par rapport à UML, diagrammes de contraintes et diagrammes paramétriques), qui est standardisé par l'OMG [78]. EAST-ADL étant implanté comme un profil UML2, le support de ce standard est garanti par construction.

Homogénéité de la description Les *niveaux d'abstraction* de EAST-ADL pouvant être assimilés à des *vues* UML, la description n'est pas homogène. On remarquera cependant qu'un effort est fourni concernant la conservation des propriétés entre les différents niveaux.

Possibilité d'annotation Le langage ne propose pas de possibilité d'annotations par l'architecte.

Capacité d'analyse EAST-ADL vise essentiellement à l'analyse — se reposant sur AUTOSAR en ce qui concerne la génération de code. La langage permet de couvrir des analyses comportementales, de sûreté, ainsi que de vérifier des contraintes sur le modèle. Le fait de pouvoir être décrit en UML2 et au moins pour partie en SysML ouvre la voie à l'utilisation d'un certain nombre d'outils. Les hypothèses sur le système testé (discrétisable), rendent impossibles la vérification d'hypothèses temporelles à l'étape architecturale (puisqu'un modèle d'exécution doit être fourni pour l'analyse), ou sur des exemples asynchrones non-triviaux. On notera que dans le domaine d'application de l'outil (l'industrie automobile), l'hypothèse synchrone est pertinente.

2.3 Approches pour l'optimisation dirigée par les modèles

Dans cette section, nous présentons différentes approches pour l'optimisation des systèmes temps-réel, éventuellement embarqués. Chacune de ces approches est caractérisée selon un ensemble de critères définis par la suite. Finalement, des techniques d'optimisation plus locales, non basées sur des modèles, sont également évoquées.

2.3.1 Éléments caractéristiques

Nous décrivons ici un ensemble de critères permettant d'insérer une méthode d'optimisation dans un processus dirigé par les modèles. Sans nous limiter à des mesures de performances, nous nous intéressons essentiellement à la modularité de la méthode, qui devrait être capable de traiter les contraintes spécifiques des différents domaines que nous visons, que ce soit en termes de performances specifiques attendues, de généralité et de compatibilité avec les formalismes les plus adaptés.

Domaine d'optimisation Selon la méthode, le domaine concerné par le processus d'optimistion couvre un éventail de systèmes plus ou moins restreint.

Langage de modélisation Les méthodes d'optimisation considérées s'appuient sur un ou plusieurs langages de modélisation. Ces langages peuvent être entièrement supportés, ou en partie seulement.

Fonction d'évaluation La fonction d'évaluation utilisée par le processus de modélisation, et éventuellement les contraintes tierces qui doivent garder leur validité tout au long du processus d'optimisation.

Possibilité de configuration de la fonction d'évaluation La fonction d'évaluation doit prendre en compte des paramètres dépendant du domaine et de la nature du projet. Il est donc important d'offrir une méthode de définition des critères d'évaluation pour qu'un processus d'évaluation soit effectivement généralisable.

Possibilité de spécifier des contraintes De même que les critères d'évaluation doivent pouvoir être spécifiés, il doit être possible de définir des contraintes applicatives sur l'architecture, de manière à rejeter certains motifs architecturaux, généralement pour assurer la sûreté ou la faisabilité de l'architecture.

Granularité de l'optimisation La granularité de la méthode d'optimisation détermine l'importance des résultats en terme d'efficacité.

2.3.2 OSATE

OSATE [1] est un applicatif déduié à un module de Eclipse [2] pour la manipulation des modèles AADL. Nous discutons dans cette section de la démarche d'optimisation proposée par les auteurs de OSATE dans [76].

Domaine de la méthode La méthode d'optimisation présentée par les auteurs d'OSATE s'applique à l'ensemble des modèles spécifiants des systèmes répartis (multiprocesseurs) asynchrones (sans horloge globale) et utilisants tous types de communications.

Langage de modélisation OSATE permet d'optimiser des modèles AADL (première version du langage). Il utilise cependant un ensemble de propriétés non-standard, ce qui impose de réécrire un modèle pré-existant avant de l'optimiser. Ainsi, un process AADL devra être décrit comme illustré dans le listing 2.1 pour que OSATE puisse évaluer ses "besoins" en terme de ressources matérielles, où les propriétés préfixées par *SEI* ne sont pas standards.

```
process DisplayManager
 features
   DMToDisplay : port group PageReturn;
   DMToPCM : port group PageRequest;

 properties
   SEI::MIPSBudget => 1200.0 MIPS;
   SEI::RAMBudget => 150.0 KB;
   SEI::ROMBudjet => 50.0 KB;
end A_Process;
```

Listing 2.1 – OSATE : un processus optimisable

Fonction d'évaluation La fonction d'évaluation proposée prend en compte l'utilisation du processeur, l'empreinte mémoire, la consommation énergétique et l'occupation de la bande passante du bus de communication. Le critère optimisé est le nombre de processeurs nécessaires : les autres critères doivent être initialement valides, faute de quoi l'algorithme échoue à trouver un nouveau modèle.

Possibilité de configuration de la fonction d'évaluation Aucune proposition n'est faite pour que l'architecte puisse modifier la fonction d'évaluation.

Possibilité de spécifier des contraintes Aucune proposition n'est faite pour que l'architecte puisse ajouter des critères de validité.

Granularité de l'optimisation La granularité du processus d'optimisation est celle du processus, puisqu'il s'agit de trouver un placement optimal des processus (composants logiciels) sur les processeurs (composants matériels). La technique d'optimisation utilisée est celle du *bin-packing* [37].

2.3.3 ArcheOpterix

Nous discutons dans cette section de l'outil ArcheOpterix, un module Eclipse qui peut être intégré directement à OSATE. Les auteurs présentent en [15] ses principes de fonctionnement.

Domaine d'optimisation Comme pour OSATE, le domaine est celui des systèmes répartis asynchrones.

Langage de modélisation Comme OSATE, ArcheOpterix prend en entrée des modèles AADL (première version), et ajoute un certain nombre de propriétés non-standards. Il est donc nécessaire de modifier manuellement un modèle pré-existant avant évaluation.

Fonction d'évaluation La fonction d'évaluation proposée prend en compte deux éléments : la probabilité de défaillance de l'application et la latence des connexions entre les processus. ArcheOpterix optimise donc le temps de réponse bout-en-bout des tâches de l'application, en minimisant les chances de défaillance dues au connexions. La problématique de la combinaison linéaire est résolue par l'utilisation d'algorithme génétiques, qui permettent une optimisation multi-critères.

Possibilité de configuration de la fonction d'évaluation Les fonctions d'évaluation peuvent être intégrées par un mécanisme d'enregistrement et d'héritage. Toutrefois, il reste difficile de décrire les fonctions d'évaluation.

Possibilité de spécifier des contraintes de validité Aucune proposition n'est faite pour que l'architecte puisse ajouter des critères de validité.

Granularité de l'optimisation Comme pour OSATE, ArcheOpterix cherche un placement proche de l'optimal, et a donc une granularité de l'ordre du processus.

2.3.4 SynDex

SynDex implémente une méthodologie pour la conception et l'optimisation des systèmes temps-réel embarqués. Nous traitons dans cette section des principes présentés par les auteurs en [85] et de façon plus approfondie en [79].

Domaine d'optimisation Le domaine d'optimisation est restreint à celui des systèmes répartis synchrones.

Langage de modélisation SynDex accepte plusieurs formalismes en entrée : le principal est nommé *graphes d'algorithme*, une notation qui lui est propre et qui porte le même nom, `SynDex`. Ce formalisme peut difficilement être considéré comme un langage de modélisation. L'outil SynDex accepte également plusieurs langages synchrones (Esterel [25], SyncCharts [16] et Signal [24]). Finalement, une passerelle vers UML/Marte est possible.
Dans notre contexte, qui est celui d'une optimisation dirigée par les modèles, seule la dernière hypothèse pourra être retenue. On notera que l'architecture matérielle doit être spécifiée dans un formalisme différent de celui utilisé pour la partie logicielle, et qu'aucune passerelle vers un langage de modélisation de haut niveau n'est encore proposé.

Fonction d'évaluation La fonction d'évaluation (implicite) utilisée par SynDex est l'ordonnançabilité, en prenant en compte le coût des communications.

Possibilité de configuration de la fonction d'évaluation Aucune proposition n'est faite pour que l'architecte puisse modifier les critères de performance.

Possibilité de spécifier des contraintes de validité Un formalisme propre à l'outil permet de définir un certain nombre de contraintes sur la topologie attendue et des contraintes de causalité entre les segments de code. Il s'agit cependant d'un formalisme au pouvoir d'expression faible au regard du code généré, puisque sa vocation est essentiellement de corriger certaines dérives de l'algorithme d'optimisation choisi, en limitant les chemins d'exécution possibles.

Granularité de l'optimisation L'optimisation est faite au niveau de l'instruction langage, et est donc très fine comparativement aux précédentes méthodes présentées. L'algorithme utilisé relève d'une approche gloutonne. L'approche semble être particulièrement efficace pour exploiter le parallélisme inhérent à l'application optimisée. On notera cependant que l'optimisation est effectuée sur le code généré, et non sur le modèle. Il y a donc génération de code partiellement non documenté, ou au moins documenté dans des formalismes hétérogènes (modèle d'architecture, d'algorithme, de déploiement...)

2.3.5 Physical Assembly Mapper

L'outil PAM [20] (Physical Assembly Maker) est un outil visant le domaine de l'avionique, dans le cadre du projet Pollux, qui est soutenu par des acteurs majeurs de l'avionique militaire des USA (Air Force Research Laboratory's Information Directorate, Lockheed Martin et Raytheon). Il vise à optimiser les applications suivant une approche par composants. Nous présentons ici cette technique.

Domaine d'optimisation Le domaine d'optimisation est restreint à celui des composants répartis. Le contexte est donc asynchrone.

Langage de modélisation PAM prend en entrée des modèles Lightweight CCM.

Fonction d'évaluation La fonction d'évaluation utilisée par PAM ne prend en compte que l'empreinte mémoire finale de l'application générée, et donc n'effectue d'optimisation que sur ce critère.

Possibilité de configuration de la fonction d'évaluation Aucune proposition n'est faite pour que l'architecte puisse modifier les critères de performance.

Possibilité de spécifier des contraintes de validité Aucune proposition n'est faite pour que l'architecte puisse ajouter des contraintes sur les modèles optimisés.

Granularité de l'optimisation Le niveau de granularité est celui du composant, soit un niveau plus fin que celui du process. L'opération d'optimisation utilisée est la *fusion* qui réunie deux composants en un seul. On notera que les bons résultats obtenus sont à comparer avec un système construit sans composant, puisque l'utilisation de ce type de modèle implique un surcoût en terme de mémoire.

2.3.6 Approches partielles

Nous traitons dans cette rubriques de techniques d'optimisation locale, qui ne sont pas forcément basées sur une approche dirigée par les modèles.

Optimisation des buffers

Les auteurs de [74] proposent un algorithme pour l'optimisation de l'accès au buffer dans des modèles Simulink [80]. Cette approche s'applique aux systèmes synchrones monoprocesseur, et n'est donc pas applicable en l'état dans le cadre de notre étude. Aucun moyen n'est donné pour spécifier une fonction d'évaluation différente ou des critères supplémentaires sur le code généré. Les bonnes performances obtenues doivent cependant servir de mesure à une étude sur l'évaluation.

Découpage et déploiement de l'application

Les auteurs de [46] proposent une technique de compilation pour générer des programmes *C* efficaces à partir de modèles Esterel. Le contexte est celui des systèmes synchrones sur architectures parallèles. Bien que le contexte soit différent, utiliser une telle approche pour le déploiement des process est envisageable.

2.4 Méthodes de vérification des contraintes

La section 2.3 décrivait plusieurs approches existantes pour l'utilisation et la transformation des modèles architecturaux en vue d'optimiser les systèmes temps-réel embarqués. Nous avons vu dans le chapitre 1 que l'optimisation entrainait un besoin de vérification de contrainte, besoin qui peut être comblé par un langage de spécification de contraintes sur le modèle.

Nous étudions donc dans cette section les différents langages pour l'expression de contrainte, et leur support en terme d'outillage.

2.4.1 Éléments caractéristiques

Expressivité Un langage d'expression de contrainte peut être général ou spécialisé pour un modèle particulier. De sa généralité dépendra son expressivité.

Capacité de raffinement Cependant, le modèle utilisé pour vérifier des propriétés n'est pas nécessairement le modèle final. En effet, on veut pouvoir vérifier des propriétés à un stade précoce du développement, avant que le modèle final ne soit établi — par exemple parce que toutes les contraintes ne sont pas encore connues. Le raffinement permet de déduire un modèle d'un autre, sans le réécrire entièrement, et surtout sans invalider les propriétés vérifiées.

Composition La composition d'un système met en œuvre différentes équipes travaillant séparément dont les travaux sont ensuite assemblés pour construire le système final. Pouvoir créer de façon indépendante, les modèles des différents composants du système permet de pouvoir les assembler pour analyse.

Automatisation Certaines méthodes formelles offrent la possibilité de procéder à des analyses de façon automatique (par model-checking par exemple) ; d'autres nécessitent l'intervention d'experts. Dans l'optique de la réalisation d'un processus d'optimisation automatisé, un tel critère est crucial.

Types de propriétés Les méthodes d'analyse ciblent des propriétés différentes. De façon non exhaustive, s'y trouvent :

- des analyses comportementales ;
- des analyses temporelles (par exemple la latence ou le temps de réponse) ;
- des analyses de dimensionnement de ressources autre que le temps processeur (énergie, mémoire, etc.).

Insertion dans le cycle de développement Enfin, il est important de pouvoir statuer sur la place que l'utilisation d'une méthode formelle peut occuper dans un processus de développement :

- en amont : ces méthodes sont utiles pour clarifier les exigences et les contraintes, pour spécifier l'architecture du système ;
- tout au long du processus : ces méthodes seront utiles pour s'assurer de la non régression du système au fur et à mesure de son développement (typiquement en permettant d'intégrer facilement de nouvelles informations dans le modèle formel) ;
- en aval : elles permettent de s'assurer que l'on respecte bien les contraintes émises en amont du processus.

2.4.2 Object Constraint Language

Object Constraint Language (OCL) est un langage pour exprimer des contraintes. S'il est généralement utilisé sur des modèles UML, le standard spécifie que son pouvoir d'expression s'applique à n'importe quel langage dont le méta-modèle a été décrit selon le formalisme OMG Meta Object Facility [77] (MOF).

Expressivité OCL est un langage extrêmement général, puisqu'il n'est lié à aucun langage en particulier, si ce n'est le méta-modèle MOF de UML. De ce fait, son expressivité est très grande. On notera cependant que cette expressivité nuit à l'apprentissage du langage - concrètement la difficulté principale dans l'écriture d'une contrainte est en effet la connaissance du ou des méta-modèles utilisés (si plusieurs vues du modèle sont concernées).

Capacité de raffinement OCL offre de bonnes capacités de raffinement.

Composition Les contraintes OCL se composent très bien, puisqu'on interroge le méta-modèle (typiquement avec des requêtes du type "toutes les instances d'une classe A doivent vérifier une propriété B").

Automatisation OCL pour UML est implanté par plusieurs outils, dont [43].

Types de propriétés Les types de propriétés vérifiables par un langage de description de contrainte sont principalement temporelles (ordonnancement, latence, etc.) et de dimensionnement (adéquation matériel/logiciel).

Insertion dans le cycle de développement OCL s'insère à tous les niveaux du cycle de développement.

2.4.3 Les approches par preuve de théorème

La méthode *Z* [13] et la méthode *B* [45] sont deux exemples de langages utilisés par des prouveurs de théorèmes — même si dans le cas du *B* il peut également être utilisé pour générer le squelette du code-source, assurant ainsi une certaine cohérence entre les différentes étapes de développement. Les deux méthodes consistent à écrire une spécification formelle du système et à exprimer des contraintes sur cette spécification. Les théorèmes à prouver sont exprimés à l'aide d'éléments de logique (souvent du premier ordre).
En combinant les différents prédicats exprimés dans le formalisme considéré, l'utilisateur doit alors être en mesure de prouver des expressions qui garantissent le respect de propriétés dans le système.

Expressivité L'expressivité des théorèmes est celle de la spécification formelle, qui — au moins dans le cas du *B* — est relativement importante, puisqu'elle permet de générer du code après des étapes de raffinement successifs [63]. Cependant, cette expressivité se limite aux aspects fonctionnels de l'application, et non aux aspects de dimensionnement ou d'échéances. Dans le contexte de systèmes TRE, ces questions ne peuvent pas être ignorées.

Capacité de raffinement La modification d'un élément existant de la spécification du système implique :

- de prouver que le nouvel élément ne présente pas d'erreur ;
- de prouver qu'il s'intègre bien dans la spécification (cohérence) ;
- de prouver que tous les éléments de preuve utilisant (dépendant de) ce nouvel élément (modifié) restent cohérents et corrects.

L'ajout d'éléments dans la spécification ne remet pas en cause les résultats déjà obtenus.

Composition Il est tout-à-fait possible, avec ce type d'approche, de spécifier dans des fichiers séparés les éléments du système. Chaque fichier peut contenir les preuves relatives à chaque élément. Ces éléments peuvent être regroupés pour produire une spécification globale, où il sera cependant nécessaire de prouver que les éléments ajoutés n'ont pas d'impact fautif sur d'autres. En outre, les contraintes peuvent être exprimées au niveau global de la spécification, puis analysées.

Automatisation Dans le meilleur des cas, l'outil d'analyse utilisé, un prouveur, est semi-automatique : l'avantage par rapport à la preuve manuelle est que le prouveur semi-automatique dispose de bibliothèques de théorèmes, de lemmes ou d'axiomes qui peuvent être intégrés à la résolution de la preuve en cours. Il propose aussi des méthodes de réduction de prédicats pour arriver au but, comme la réduction *one-point rule* qui permet de traiter les quantificateurs dans les prédicats.

Types de propriétés Les approches par preuve de théorèmes permettent d'analyser différents types de propriétés sur une spécification donnée :

- vérification de type ;
- validité du domaine d'application de fonctions ou d'opérations ;
- cohérence des interactions entre composants (typage, paramètres).

Insertion dans le cycle de développement Les prouveurs de théorèmes sont essentiellement utilisés au début du processus de développement pour clarifier les exigences fonctionnelles, fixer les interfaces et établir une architecture cohérente. Ils peuvent ensuite être utilisés pour générer du code.

2.4.4 Les approches par supervision d'exécution

L'approche par supervision d'exécution implique d'étudier les retours d'exécution de l'application pour adapter dynamiquement les paramètres de celle-ci de manière à la rendre exécutable en fonction des contraintes (ordonnançabilité, mémoire disponible, etc.). L'intérêt principal d'une telle approche est qu'elle n'exige pas une connaissance a priori des caractéristiques réelles du système, mais uniquement des contraintes qui s'exercent sur celui-ci. Les auteurs de [14] présentent un résumé de cette approche dans les systèmes informatiques. Dans [70], une technique pour adapter cette approche aux systèmes TRE est proposée, ainsi qu'une condition sur la faisabilité et la contrôlabilité de ces systèmes.

Expressivité Il n'existe pas à notre connaissance de mécanisme d'expression des contraintes pour les systèmes superviseurs. Ces systèmes manquent donc de flexibilité dans leur usage, puisqu'ils ne peuvent vérifier qu'une propriété donnée.

Capacité de raffinement Les systèmes de supervision ne faisant pas d'hypothèses fortes sur l'architecture logicielle de l'application, le raffinement ne pose pas de problème particulier.

Composition La composition de systèmes TRE *supervisables* (faisables et contrôlables), par exemple dans un système distribué, n'est pas nécessairement supervisable. C'est le cas en particulier si existe des dépendances entre les tâches des systèmes composés. Pour résoudre ce problème, une condition rendant cette composition possible est présentée par les auteurs de [89]. Celle-ci demande cependant de connaitre l'ensemble des systèmes participants à la composition : c'est-à-dire que l'analye doit être entièrement refaite dans ce cas.

Automatisation les systèmes RTEMS [4] et VxWorks [5] fournissent des plate-formes de supervision pour les systèmes TRE.

Types de propriétés La supervision permet de surveiller l'utilisation des ressources matérielles ou logicielles, donc essentiellement l'ordonnancement et l'empreinte mémoire.

Insertion dans le cycle de développement Si l'existence du superviseur doit être intégrée dès la phase de conception, il ne fournira des informations que durant les tests d'intégration — la réussite des tests unitaires n'étant pas gage de faisabilité après composition, comme vu précédement.

2.5 Positionnement

Pour notre approche, il nous fallait déterminer un ADL, puis une méthode pour exprimer des contraintes, méthode qui doit être accessible à un architecte non-expert, et enfin trouver des techniques d'optimisation qui puissent s'appliquer efficacement dans ce contexte.

2.5.1 ADL

Nous avons vu que peu d'ADLs répondaient de façon native aux contraintes que nous avions fixées, c'est-à-dire un langage standardisé, homogène, offrant des possibilités d'annotation et outillé pour l'analyse. Il est possible de modifier les langages pour combler ces lacunes, mais pour qu'une méthode soit généralisée, il faut qu'elle s'appuie sur un langage standard, sous peine de générer d'importants problèmes de maintenance. Dans ces conditions, seuls AADL et UML/MARTE restent en lice. Nous avons choisi de sélectionner AADL comme langage-pivot pour le développement — et donc pour l'optimisation — et ce pour plusieurs raisons :

- AADL ne comporte qu'une seule vue du modèle, ce qui permet d'éviter les problèmes d'incohérence ;
- AADL est défini sur un méta-modèle beaucoup plus simple que MARTE — lequel dérive du méta-modèle UML — et par conséquent un modèle AADL peut être analysé de façon beaucoup plus efficace en terme de temps processeur et surtout d'empreinte mémoire. Dans le cadre d'une optimisation où un grand nombre d'instances peuvent coexister simultanément en mémoire, un tel critère est primordial ;
- AADL est standardisé depuis plus longtemps, et est plus outillé que MARTE. Il promet donc des possibilités d'analyse supérieures, au moins dans un futur proche.

Une étude [75] effectuée par les auteurs de AADL corrobore notre choix : les auteurs y présentent une comparaison entre AADL et UML, indiquant l'intérêt d'utiliser AADL comme language de description d'architecture et de restreindre l'usage d'UML (et de son profil MARTE) à un langage de description de la structure fonctionnelle.

De plus, MARTE est plus vaste que AADL en terme d'informations couvertes, et d'un certain point de vue contient celui-ci. Or, couvrir une vaste gamme d'informations n'est pas forcément une approche pertinente pour une modélisation efficace, puisqu'elle implique un accroissement correspondant de la phase d'analyse. Le comité MARTE a d'ailleurs spécifié un document d'annexe pour décrire les patrons de modélisation permettant de réaliser un modèle MARTE à partir d'un modèle AADL, dans le but d'intégrer une passerelle entre les deux formalismes.

2.5.2 Évaluation des performances et des contraintes

Nous voulons également disposer d'un langage permettant aussi bien à l'architecte qu'aux auteurs du processus d'optimisation d'exprimer des fonctions d'évaluation directement sur le modèle original, sans passer par une représentation tierce du modèle.

Les méthodes formelles que sont les prouveurs de théorème et le model-checking (par exemple les réseaux de Petri) ne correspondent pas à cette définition, car ils imposent de fournir une spécification formelle du modèle original. Quant aux approches par supervision d'exécution, elles ne permettent pas à l'heure actuelle d'exprimer une contrainte autrement que dans le code source. De plus, l'aspect dynamique de ces dernières les rend impropres à l'analyse des systèmes critiques. Une approche semie-formelle, par description de contraintes, répondrait à notre besoin.

Cependant, l'évaluation d'OCL nous pose plusieurs problèmes :

- la trop grande expressivité du langage le rend difficilement maîtrisable ;
- OCL pour UML nécessite que l'architecte connaisse le méta-modèle du langage de modélisation, et non simplement le langage lui-même.

Ce dernier point, plus particulièrement, nous semble bloquant. Notre approche se voulait basée sur un langage pivot unique. Idéalement, l'architecte ne devrait pas différencier la définition du modèle de celle des contraintes — de façon analogue à ce qui se passe dans les systèmes de gestion de bases de données, où le passage entre le langage de description de données et le langage de recherche et de requête est généralement transparent à l'utilisateur.

Nous avons donc opté pour la réalisation d'un langage de description de contrainte qui nous permette de combler ces deux manques de OCL. Ce langage, REAL, est décrit dans le chapitre 5.

2.5.3 Processus d'optimisation

Nous nous proposons de définir un processus qui permette l'optimisation des systèmes TRE par l'ingénierie des modèles. Pour cela, nous avons besoin d'un processus répondant aux critères suivants :

1. basé sur un langage de modélisation de haut-niveau, que ce soit en entrée ou en sortie du processus s'appliquant aux systèmes embarqués temps-réel répartis asynchrones ;
2. appliquant une optimisation à granularité fine ;
3. permettant des évaluations de performances et de contraintes avant, pendant et après le processus d'optimisation ;
4. offrant des fonctions d'évaluations qui puissent être modifiables par l'architecte.

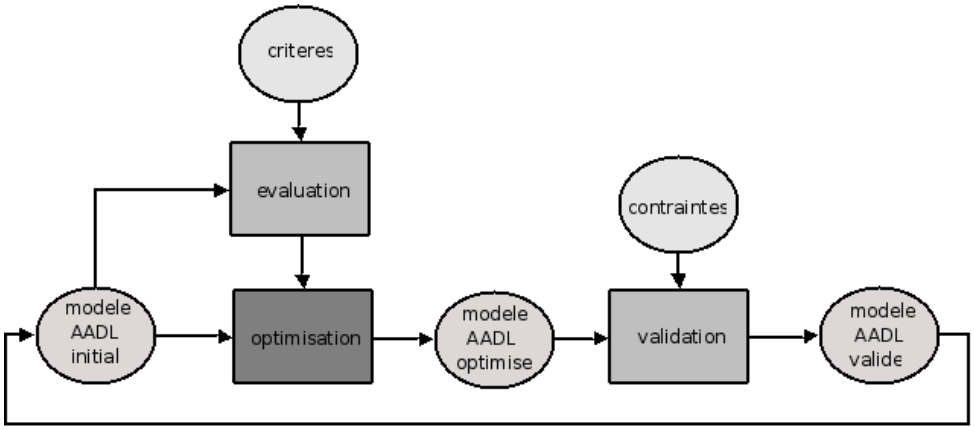


FIG. 2.3 – Les différentes étapes de l'optimisation

A partir des exigences que nous venons de citer, nous pouvons déduire le schéma 2.3. Les ellipses gris médian représentent les différentes étapes du modèle AADL (point 1), tandis que le rectangle gris foncé illustre l'étape d'optimisation (point 2, traité dans les chapitres 3 et 4). Les rectangles gris médian représentent quant à eux les étapes d'évaluation et de validation (point 3, traité dans les chapitres 4 et 6). Finalement, les ellipses gris clair illustrent des contraintes ou critères spécifiés par l'utilisateur ou définis par défaut, dans un formalisme d'apprentissage rapide (point 4, traité dans les chapitres 5 et 6).

3

Optimisation des systèmes Temp-Reél Embarqués

Nous avons établi au chapitre 2 que l'optimisation d'un système TRE nécessitait une vision complète de l'architecture d'un système, et que la modélisation via un langage de description d'architecture fournissait un bon niveau d'abstraction.
Dans ce chapitre, nous discutons plusieurs techniques d'optimisation en nous basant sur un modèle de l'architecture. Nous évaluerons par la suite les techniques basées sur le code source.

Toute tentative de changer les caractéristiques non-fonctionnelles ou topologiques d'un système en modifiant son modèle implique de comprendre la relation qui existe entre ce modèle et le système généré. Dans ce chapitre, nous resterons génériques et exprimerons cette relation dans le cadre d'un intergiciel minimal supposé capable de gérer l'ensemble de l'application répartie, à l'exclusion de la partie fonctionnelle. Nous verrons comment, à partir d'une analyse de l'étude de l'impact de transformation du code source, nous pouvons définir des opérations d'optimisation sur les modèles.

3.1 Relation modèle-performances

La relation modèle-performance est conditionnée par deux aspects : d'une part le comportement temporel de l'application, qui est explicite dans le modèle, et d'autre part le code généré par l'intergiciel, qui peut être connu et potentiellement dépendant du modèle.

Dans cette section, nous nous baserons sur le formalisme AADL (`threads`, `processes`, `systems`...) pour décrire les différentes éléments de notre application, et indiquer comment les modifications de ces éléments peuvent avoir un impact sur les performances de l'application.

3.1.1 Modification du comportement temporel

Le comportement temporel de l'application est directement déterminé par l'agencement du modèle. A l'intérieur d'un `process`, chaque `thread` est une tâche dont l'exécution s'exécute en concurrence des autres, sans toutefois pouvoir être en parallèle puiqu'ils s'exécutent sur le même processeur. Dans le cadre de notre étude, nous limitons notre étude au cas des systèmes ordonnançables avec Rate Monotonic Analysis (RMA), car celui-ci est optimal pour la classe des algorithme à priorité fixe, et que le déterminisme de son comportement correspond aux besoins courants des systèmes critiques que sont généralement les systèmes temps-réel embarqués — c'est pourquoi il s'agit de l'algorithme d'ordonnancement le plus communément supporté

par les différents systèmes d'exploitation temps-réel. Un thread peut être déclenché soit par l'horloge système (il sera alors dit périodique), soit par l'arrivée d'un évènement (il sera alors dit apériodique ou sporadique, selon qu'il puisse être exécuté un nombre infini de fois sur un intervalle ou non). On notera que les communications étant asynchrones (c'est-à-dire non-bloquantes), les tâches sont indépendantes. Conformément aux hypothèses nécessaires à RMA, nous imposons un intervalle minimum entre la réception de deux instances d'un même signal pour un thread non-périodique (nous n'autorisons donc pas les thread apériodiques, mais uniquement les threads périodiques et sporadiques). Les différents `process` sont chacun associés à un processeur différent, et exécutent donc leurs threads en parallèle.

En modifiant l'agencement du modèle, nous pouvons modifier son comportement temporel, et donc les performances. Dans cette section, nous distinguons plusieurs types d'opérations sur le modèle et nous montrons leur impact sur les performances du système.

Sérialisation

La sérialisation consiste à agréger des parties concurrentes de code pour les exécuter séquentiellement. Si le code aggrégé était initialement exécuté en parallèle, une baisse du débit peut être attendue, mais également une baisse du nombre de communications. On sait que ces dernières sont déterminantes dans l'évaluation du pire temps d'exécution, ainsi qu'un facteur majeur de complexité et d'erreurs potentielles.

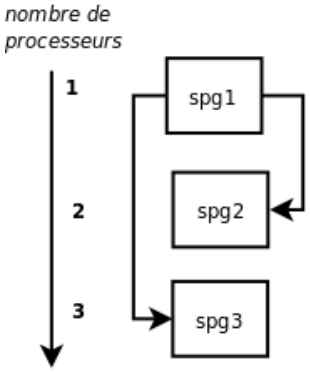


FIG. 3.1 – Sous-programmes sérialisables

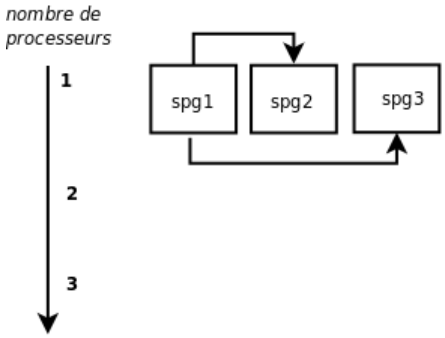


FIG. 3.2 – Sous-programmes sérialisés

La figure 3.1 illustre un cas où une opération de sérialisation entre trois sous-programmes exécutés en concurence est pertinente. Dans la figure, les sous-programmes *spg*2 et *spg*3 dépendent de *spg*1 — ils devront donc attendre la fin de son exécution dans une implantation concurrente. Le parallélisme effectif sera donc limité à *spg*2 et *spg*3. A contrario, la sérialisation de ces trois sous-programmes (illustrée par la figure 3.2) permet de libérer deux connexions et deux threads, et donc les ressources matérielles associées.

Parallélisation

La parallélisation consiste à séparer des parties de code séquentielles pour les exécuter sur différents processeurs. Une application extrême d'une telle procédure est le pipelining, qui permet d'assurer un débit maximal. Cette approche provoque cependant une augmentation importante du nombre de communications inter-processeurs, et donc de temps de latence faiblement maîtrisées. Un autre problème lié à cette approche est le risque plus important de panne matérielle.

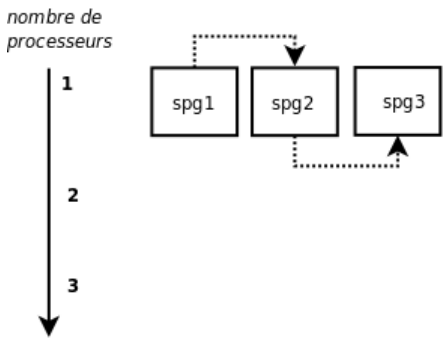


FIG. 3.3 – Sous-programmes parallélisables

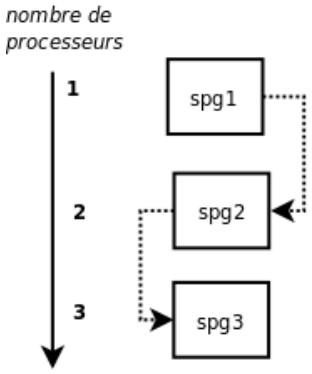


FIG. 3.4 – Sous-programmes parallélisés

L'opération de parallélisation est illustrée par les figures 3.3 et 3.4, qui montrent trois sous-programmes initialement séquentiels qui sont parallélisés. Dans cet exemple, les sous-programmes partagent des connexions retardées (chaque consommateur a en entrée les données produites par ses producteurs lors de leur cycle précédent). La latence du système est conservée — puisque chaque donnée doit être traitée par les trois sous-programmes — mais la parallélisation permet d'effectuer du pipelining, c'est-à-dire que si $WCET(s)$ est le pire temps nécessaire pour exécuter le sous-programme s, une donnée pourra, en moyenne, être traitée dans un temps égal à :

$$Max(WCET(spg1), WCET(spg2), WCET(spg3))$$

(contre

$$WCET(spg1) + WCET(spg2) + WCET(spg3)$$

pour traiter une donnée dans la version séquentielle).

Équilibrage de charge

Dans le cas de systèmes multi-processeurs (ou de processeurs multi-cœurs), l'équilibrage consiste à effectuer un placement des composants logiciels actifs sur les différents processeurs en fonction de la charge pré-existante, de manière à égaliser la charge finale sur chaque processeur. Dans l'hypothèse d'un placement optimal, on peut attendre une baisse de l'occupation maximale des processeurs et donc une baisse possible de leur fréquence - ou un baisse de la taille maximale de l'empreinte mémoire. Nous noterons que nous nous restreignons au cadre d'un équilibrage statique, pour préserver le déterminisme de l'application.

La figure 3.5 illustre le cas de trois sous-programmes séquentiels, sur lesquels une opération d'équilibrage peut être appliquée. Deux de ces sous-programmes sont étoitement interconnectés (par une connexion directe dans un sens et par une connexion retardée dans l'autre sens). Le troisième sous-programme n'est cependant pas connecté aux deux autres. Sous réserve que le temps d'exécution de ce troisième sous-programme ne soit pas négligeable par rapport aux deux précédents, une opération d'équilibrage de charge doit être effectuée, et le *spg*3 doit être déplacé sur un nouveau processeur en sous-charge, comme illustré dans la figure 3.6.

3.1.2 Nature du code généré

Nous avons vu dans le chapitre 2 que notre processus reposait sur la génération de code pour assurer la cohérence entre le modèle et son implantation. Si la modification du comportement

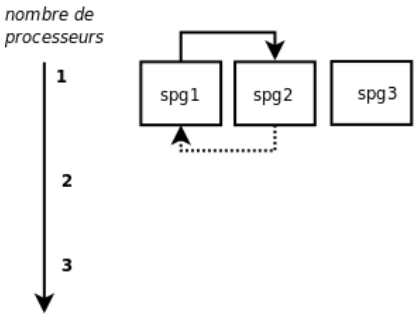


FIG. 3.5 – Équilibrage : exemple initial

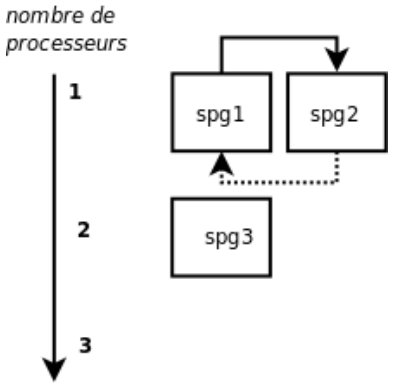


FIG. 3.6 – Équilibrage : déplacement de sous-programme

temporel d'une application permet de déterminer une certain nombre d'optimisations possibles, la connaissance des motifs de génération de code nous permet d'en déterminer quelques autres, neutres du point de vue comportemental.

Les hypothèses de génération de code retenues sont décrites précisément dans les travaux de Béchir Zalila [91], qui expose les principes de la génération de code utilisés par Ocarina [61], et de l'intergiciel sous-jacent PolyORB-HI [92], générateur de code avec lequel nous avons travaillé durant cette étude. Nous décrirons, dans cette section, les éléments de cette opération qui dépendent du modèle et ont un impact sur les performances. Dans la suite de ce chapitre, nous décrirons également les modifications que nous avons apportées à cette opération.

Couche de composants logiciels

Tout intergiciel va construire une sur-couche consommatrice de ressources matérielles au-dessus du code-métier — essentiellement en terme de mémoire, mais potentiellement de temps processeur également. Cette sur-couche s'appliquera au minimum aux composants actifs (`threads`), mais peut toucher également certaines données ou procédures. Dans le cas de notre intergiciel minimum théorique, nous ferons l'hypothèse que seuls les threads vont générer une sur-couche composée uniquement d'un accroissement de l'empreinte mémoire. Il en découle que la suppression d'un composant actif permet de réduire l'empreinte mémoire de l'application.

Communications

A notre échelle qui est celle de systèmes embarqués de taille intermédiaire à supérieure (c'est-à-dire disposant d'une mémoire au moins égale à 100 Ko), nous considérons que nous travaillons sur des systèmes d'exploitation à part entière, et que l'application générée sera en concurrence avec d'autres applications pour l'accès aux différentes ressources. C'est pourquoi les connexions, y compris celles qui sont locales ont un accès protégé. Comme l'émetteur et le récepteur sont typiquement de période différente et que les communications sont asynchrones, la gestion des communications doit impliquer une file d'attente et un moyen de synchronisation côté récepteur. Ces éléments vont induire un surcoût faible en terme de mémoire, mais vont également provoquer une estimation pessimiste du pire temps d'exécution. Or, dans un contexte d'optimisation hors-ligne, c'est sur cette évaluation que repose l'efficacité des méthodes d'optimisation. Supprimer des connexions inter-thread - et a fortiori inter-process - permet donc d'effectuer une analyse moins

pessimiste du temps de traitement de bout en bout, réduisant en conséquence la complexité de la topologie du système.

3.2 Opérations sur les modèles

Les techniques décrites ci-dessus peuvent se diviser en deux types : recombinaisons et factorisations de composants. Pour pouvoir explorer l'espace des solutions possibles, nous ajoutons par ailleurs une opération de division. Dans le modèle, ces opérations se manifestent par trois procédures :

- `merge`, qui factorise deux threads co-localisés dans un même process en un seul ;
- `move`, qui déplace un thread d'un process à un autre ;
- `split`, qui divise un thread en plusieurs threads.

On notera que déplacer une partie du contenu d'un thread vers un autre (l'équivalent d'un move sur les threads) peut être fait par utilisation successive d'un split et d'un merge. Dans la suite de cette section, nous commençons par donner une description des critères pertinents pour définir une opération donnée puis nous montrons, pour chaque opération, comment la mener à bien et quelles propriétés on peut attendre du système transformé, avant de conclure sur les opérations que nous pourrions ajouter à cet ensemble.

3.2.1 Structure du problème

Toute opération sur les modèles architecturaux consiste à modifier les caractéristiques de ces modèles, telles que décrites dans le chapitre 2. Il est donc naturel que l'on retrouve la classification des éléments distinctifs des langages architecturaux dans leur description :

- modifications des interfaces ;
- modifications topologiques ;
- modifications des annotations ;
- modifications comportementales.

Par ailleurs, modifier les modèles implique de construire de nouveaux composants. Les identifiants de ces nouveaux composants doivent pouvoir assure leur tracabilité, et garantir de ne pas entrer en collision avec les identifiants existants. C'est pourquoi nous aurons également besoin de *règles de nommage*.

De plus, on peut définir des conditions de faisabilité sur une opération. Ces conditions peuvent être intrinsèques à la nature même de l'opération — auquel cas on parlera de *précondition*, ou la conséquence de sa réalisation — auquel cas on parlera de *postcondition*. Dans les sections suivantes, nous distinguerons ces deux types de contraintes en faisant précéder la description des étapes de l'opération par la première, et en la faisant suivre par la seconde.

3.2.2 Définition du modèle initial

Pour décrire certaines transformations, nous avons besoin de définir le comportement d'un modèle de façon plus formelle que ne le fait la syntaxe AADL. Dans cette partie, nous décrivons les principaux éléments du modèle pertinent dans le cadre de l'évaluation des effets d'une opération.

Call sequence

AADL décrit un contenant pour un ensemble d'appels de sous-programmes nommé `call sequence`. Un thread (mais également un sous-programme) ne peut appeler qu'une seule `call`

`sequence` pour une configuration donnée. La configuration est représentée par un `mode` qui peut changer dynamiquement. Dans notre modélisation du problème, nous encapsulons tous les appels de sous-programmes d'une `call sequence` dans un sous-programme unique, de sorte que toute call sequence ne comporte au final qu'un seul sous-programme. Ce sous-programme est utilisé pour conserver un certain nombre d'informations qui sont propre à sa `call sequence`. Nous ne faisons cependant pas cette hypothèse sur le modèle d'origine.

Nous divisons une `call sequence` en deux aspects : les aspects fonctionnels et les aspects temporels. les aspects fonctionnels de la `call sequence` sont définis par $CS_F = (I_{CS}, O_{CS}, (SP)_i)$, où I_{CS} sont les ports d'entrée du thread utilisés par la `call sequence`, O_{CS} sont les ports de sortie, et (SP) la notation pour la suite de sous-programmes à exécuter. Les aspects temporels sont définis par $CS_T = (T, I, W, P, S)$ où :

$$CS_T : \begin{cases} T & \text{est un évènement déclencheur, soit par l'horloge ;} \\ & \text{soit par un évènement envoyé par une autre entité ;} \\ I & \text{Intervalle minimum de déclenchement (ou période) ;} \\ W & \text{Pire cas d'exécution (WCET) ;} \\ P & \text{Priorité ;} \\ S & \text{Une fonction projetant l'ensemble des entrées} \\ & I_{CS} \text{ vers les sorties } O_{CS}. \end{cases}$$

Une call sequence est dite périodique si son déclenchement est causé par l'horloge, et sporadique dans le cas contraire.

Thread

Un *thread*, quant à lui, est défini par

$$\mathcal{T} = ((CS_T)_T, \mathcal{I}, P, S_I, S_O)$$

où $(CS_T)_T$ est l'ensemble des call sequences exécutées par $\mathcal{T}$, $\mathcal{I}$ l'intervalle minimum entre deux exécutions et P sa priorité, S_I et S_O deux fonctions projetant respectivement l'ensemble des entrés et des sorties de $\mathcal{T}$ vers I_{CS_T} (respectivement O_{CS_T}).

On notera que la condition

$$\mathcal{I} = PGCD(\{I_{CS} | CS \in CS_{\mathcal{T}}\})$$

doit toujours être vérifiée. Celle-ci indique que la période du thread est égale au plus grand dénominateur commun des call sequences qui le composent. Cette politique permet, durant une itération du thread, d'exécuter n'importe laquelle de ses call sequences.

Un thread est dit sporadique si toutes ses call sequences sont sporadiques, périodique si toutes ses call sequences sont périodiques, et hybride dans les autres cas.

Process

Un *process* étant un ensemble de threads, nous le définissons comme suit :

$$\mathcal{P} = (T_P, S_I, S_O)_P$$

où $(T_P)_P$ est l'ensemble des threads contenus par $\mathcal{P}$, S_I et S_O deux fonctions projetant respectivement l'ensemble des entrés et des sorties de T_P vers les entrées (respectivement les sorties) de $\mathcal{P}$.

System

Finalement, un *system* est un ensemble de process défini par :

$$\mathcal{S} = (P_S, S_I, S_O)_s$$

où $(P_s)_s$ est l'ensemble des process contenus par $\mathcal{S}$, S_I et S_O deux fonctions projetant respectivement l'ensemble des entrés et des sorties de P_S vers les entrées (respectivement les sorties) de $\mathcal{S}$.

Composants matériels

AADL permet de définir les composants matériels qui composent la plate-forme d'exécution des divers composants matériels. Il permet également d'associer ces deux types d'entités en utilisant des propriétés standards. Une méthode d'optimisation possible est donc la problématique du placement, qui correspond typiquement à un problème de bin packing. Dans le cadre de notre travail, cependant, nous ne traitons pas de cette approche, car nous la jugeons suffisamment étudiée. La section 2.3 évoque quelques exemples de techniques de placement, que ce soit sur les processeurs ou sur les bus.

3.3 Opération Move

L'opération *Move* a deux objectifs : améliorer l'équilibre de la charge entre chaque process, et rendre accessible l'ensemble des solutions possibles pour un modèle donné.

Son effet sur les performances du système est très variable, puisqu'elle procède de l'équilibrage de charge entre les process. Dans l'hypothèse la plus simple possible — un process par processeur — on peut supposer que si la répartition de charge par processeur s'améliore, on est en droit d'attendre une réduction significative du temps de réponse des tâches. Néanmoins, si la nouvelle coupure dans le graphe qui représente la topologie du système augmente, le nombre de connexions inter-process, la consommation de ressources matérielles et les temps de latence vont augmenter - provoquant ainsi une augmentation non seulement du temps de réponse mais également du WCET (lié à la gestion des nouvelles connexions).

Finalement, dans certains cas, une utilisation efficace de cette opération peut permettre le pipelining des applications, et donc un gain significatif en terme de débit — au prix d'une augmentation de la latence et de la consommation de ressources.

Dans le cas de déplacement de threads, un certain nombre de facteurs vont influer sur la reconfiguration des connexions :

- la source de la connexion (P_{src}, P_{dst} ou processus tiers) ;
- la destination de la connexion (P_{src}, P_{dst} ou processus tiers) ;
- l'unicité de la connexion sur les différentes features qu'elle traverse.

Le cas le plus simple est celui où t_{mov} (le thread candidat au déplacement) est connecté à un thread de P_{dst} (le process destination de t_{mov}), et qu'aucune autre connexion ne passe par les features traversées par le flot. la figure 3.7 illustre ce cas dans un modèle AADL.

Le modèle résultant du `Move` est illustré par la figure 3.8. Puisque le thread t_{mov} est déplacé dans P_{dst}, nous établissons une connexion directe entre les ports des deux threads connectés. Cette opération autorise la suppression des connexions inutiles de P_{src} et P_{dst}. Il est à noter que le cas où t_{mov} est $thr2$ (et non $thr1$) est exactement symétrique du cas présenté précédemment. De la même manière, si $thr1$ et $thr2$ sont sous-composants du même process (donc reproduisant la situation du modèle final 3.8), un move sur l'un de ceux-ci ramènera à la situation originale

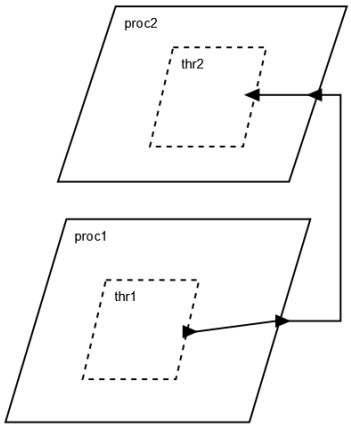


FIG. 3.7 – Connexion unique entre proc1 (P_{src}) et proc2 (P_{dst})

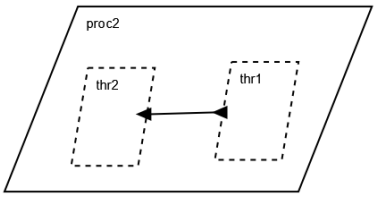


FIG. 3.8 – Connexion unique entre proc1 (P_{src}) et proc2 (P_{dst}) : Move

illustrée par la figure 3.7 — il y aura donc création d'un port dans chaque process, et d'une connexion entre ces ports.

Dans le cas du déplacement d'une connexion partageant le port d'émission avec d'autres connexions, comme la figure 3.9 l'illustre, nous ne pouvons pas supprimer le port de P_{src}, puisque celui-ci est utilisé par les autres connexions. La connexion inter-process doit également être conservée, pour la même raison. La figure 3.10 illustre le modèle fusionné correspondant à cette situation.

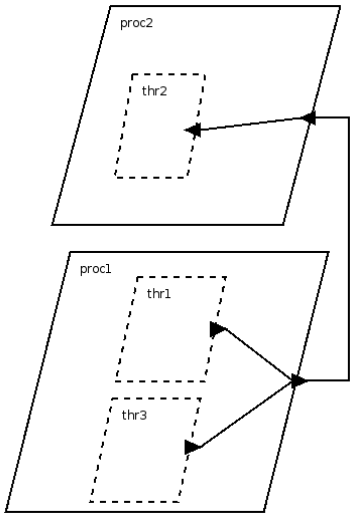


FIG. 3.9 – Connexion multiple entre proc1 (P_{src}) et proc2 (P_{dst})

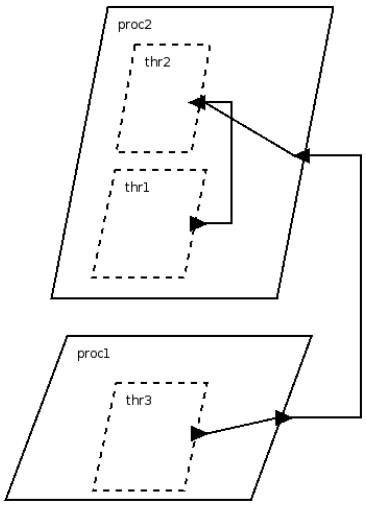


FIG. 3.10 – Connexion multiple entre proc1 (P_{src}) et proc2 (P_{dst}) : Move

Le dernier cas se présentant est celui où la source, ou la destination d'une connexion est un thread d'un process tiers. La figure 3.11 présente un exemple où t_{mov} est $thr1$, et P_{dst} est $proc2$. t_{mov} est connecté (de façon unique, et avec des ports `data`) au process tiers $proc3$. Le modèle résultant 3.12 montre que la connexion au niveau system est déplacée vers P_{dst}, tout comme les

ports. On notera que les même restrictions s'appliquent quant à la suppression des anciens ports et connexions que dans le cas des connexions locales : si la connexion entre $thr1$ et $thr2$ n'avait pas été unique (au sens défini ci-dessus), le port associé dans $proc1$ aurait été conservé, de même que la connexion elle-même.

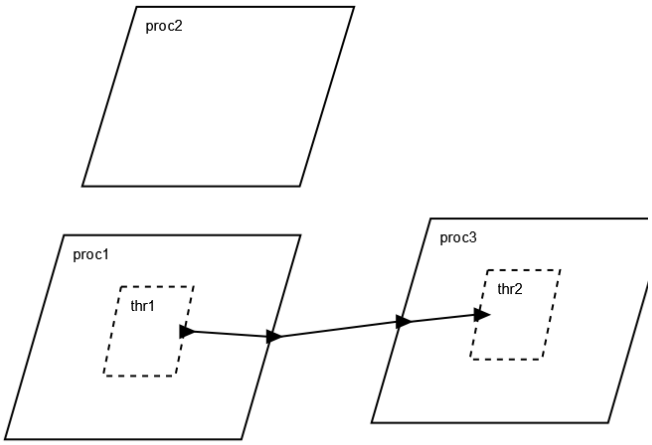


FIG. 3.11 – Connexion entre proc1 (P_{src}) et un processus tiers

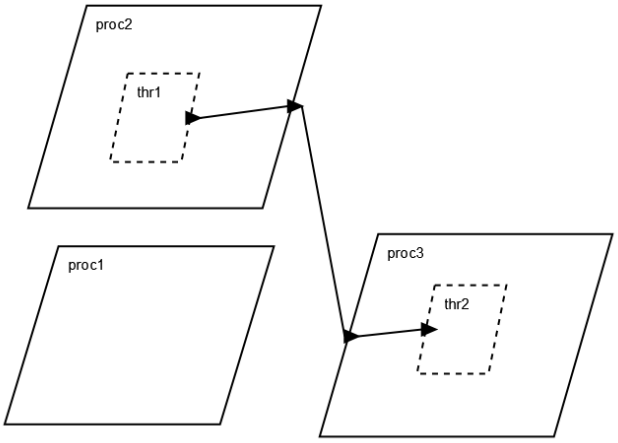


FIG. 3.12 – Connexion entre proc1 (P_{src}) et un processus tiers : Move

3.3.1 Préconditions

P_{src} et P_{dst} doivent appartenir au même system. Le thread t_{mov} ne doit pas accéder à un périphérique.

3.3.2 Règles de nommage

Nous définissons l'opération Move (Mo) sur une instance de thread et un process comme suit : $T \subset Threads, Mo(t_{mov}, P_{dst}) : P \rightarrow P$, où P est l'ensemble des process définissant un système. Dans le modèle AADL, il faut nous assurer que le nom du nouveau sous-composant de P_{dst} ne rentre pas en conflit avec un sous-composant déjà présent.

Pour ce faire, nous assignons un identifiant unique à ce sous-composant, constitué d'un préfixe désignant la catégorie du composant, les identifiants du process et du thread d'origine (pour assurer la traçabilité) et un entier unique pour cette catégorie dans l'espace de nom du process (pour éviter la collision des noms). La transformation 3.3.1 illustre ces règles.

Transformation 3.3.1 (Nommage d'un thread)

$$R_0 : identifiant \leftarrow thr_ < P_{src} > _ < t_{mov} > _ < unique > \tag{3.1}$$

où $< unique >$ *est un entier unique pour les sous-composants du process.*

Ce nouvel identifiant est ensuite ajouté aux sous-composants de P_{dst}, et l'ancien identifiant de t_{mov} est supprimé de la liste des sous-composants de P_{src}, suivant la règle 3.3.2.

Transformation 3.3.2 (Déplacement du thread)

$$R_1 : \begin{cases} T_{P_{dst}} \leftarrow T_{P_{dst}} \cup r_0(t_{mov_{src}}); \\ T_{P_{src}} \leftarrow T_{P_{src}} - t_{mov_{src}}. \end{cases}$$

Des données peuvent être spécifiées dans les sous-composants du thread t_{mov}. Dans ce cas, nous les déplaçons avec le thread, à moins qu'elles ne soient partagées, auquel cas nous appliquons la règle définie dans la section 3.3.8. Nous établissons donc la règle 3.3.3, s'appliquant à toute déclaration de sous-composant *sc* dans t_{mov}. Il doit être noté que cette dernière recouvre les éventuels objets locaux (*priority_ shifters*) créés durant une opération `Merge`, tel que décrit en 3.4.6.

Transformation 3.3.3 (Déplacement des objets locaux)

$$R_2 : subcomponents(t_{mov_{dst}}) \leftarrow subcomponents(t_{mov_{src}}) \cup sc \tag{3.2}$$

Les features peuvent également être recopiées lors de la reconfiguration des connexions. Pour éviter les collisions de noms et pour assurer la traçabilité, nous utilisons la règle 3.3.4.

Transformation 3.3.4 (Copie d'un port)

$$R_3 : \begin{cases} identifiant \leftarrow < t > _ < f > _ < unique > & si\ f\ est\ un\ port\ du\ thread\ t; \\ identifiant \leftarrow < p > _ < f > _ < unique > & si\ f\ est\ un\ port\ du\ process\ p; \end{cases}$$

où $< unique >$ *est un entier unique pour les sous-composants du process et* f *est la feature à copier.*

Des connexions doivent également être déplacées, que ce soient celles directement connectées au thread déplacé, ou celles connectées aux mêmes features de process — et donc indirectement connectées. Les cas où une telle opération est nécessaire sont spécifiés dans la section 3.4.4. Pour pouvoir l'effectuer, nous proposons la règle 3.3.5, qui permet d'assurer la traçabilité de la connexion recopiée ou déplacée.

Transformation 3.3.5 (Copie d'une connexion)

$$R_4 : \begin{cases} identifiant \leftarrow < t > _ < c > _ < unique > & Si\ c\ est\ une\ connexion\ du\ thread\ t; \\ identifiant \leftarrow < p > _ < c > _ < unique > & Si\ c\ est\ une\ connexion\ du\ process\ p; \end{cases}$$

où $< unique >$ *est un entier unique pour les sous-composants du process, et* c *est la connexion à copier.*

3.3.3 Modification des types et interfaces

L'interface des threads déplacés est toujours conservée. Nous pouvons donc exprimer la règle 3.3.6 concernant les ports du thread t_{mov} :

Transformation 3.3.6 (Copie des ports)

$$\forall f \in features(t_{mov_{src}}) R_5 : features(t_{mov_{dst}}) \leftarrow features(t_{mov_{dst}}) \cup f$$

3.3.4 Modifications de la topologie

Pour maîtriser la complexité des différentes configurations illustrées précédemment, nous définissons deux étapes dans la reconfiguration des connexions : la copie et le nettoyage. Dans les définitions suivantes, nous définissons la notion de connexion *connectée* à un composant par le fait qu'il existe au moins un flot passant par la connexion et arrivant au composant. Une connexion peut n'être ni directe, ni exclusive.

Copie La copie consiste à ajouter les nouvelles connexions induites par le Move. Pour toute connexion C appartenant à P_{src}, une reconfiguration doit être effectuée si la source ou la destination de C est t_{mov}. Nous distinguons alors trois cas possibles :

1. C est connectée à un thread de P_{src} → *Règle 3.3.7* ;
2. C est connectée à un thread de P_{dest} → *Règle 3.3.8* ;
3. C est connectée à un thread d'un process tiers (ni P_{src} ni P_{dst}) → *Règle 3.3.9*.

Transformation 3.3.7 (Reconfiguration des connexions #1)

$$\forall t_{cnt} R_6 : \begin{cases} \textit{Ajouter } P2 \leftarrow R_3(f) \textit{ à } P_{src}; & \text{– Règle 3.3.4} \\ \textit{Ajouter } P2 \leftarrow R_3(f) \textit{ à } P_{dst}; & \text{– Règle 3.3.4} \\ \textit{Ajouter } R_4(C) \textit{ entre } t_{cnt} \textit{ et } f_1 \textit{ dans } P_{src}; & \text{– Règle 3.3.5} \\ \textit{Ajouter } R_4(C) \textit{ entre } t_{mov} \textit{ et } f_2 \textit{ dans } P_{dst}; & \text{– Règle 3.3.5} \\ \textit{Ajouter } R_4(C) \textit{ entre } f_1 \textit{ et } f_2 \textit{ dans le system}; & \text{– Règle 3.3.5} \\ \textit{Supprimer } C & \\ \textit{Supprimer la connexion de } f_2 \textit{ à } t_{mov} & \end{cases}$$

Avec t_{cnt} les threads auxquels C est connectée par un port f.

Transformation 3.3.8 (Reconfiguration des connexions #2)

$$\forall t_{cnt} R_7 : \begin{cases} \textit{Ajouter } R_4(C) \textit{ entre } t_{cnt} \textit{ et } t_{mov} \textit{ dans } P_{dst}; & \text{– Règle 3.3.5} \\ \textit{Supprimer } C; & \end{cases}$$

Avec t_{cnt} les threads auquels C est connectée par un port f.

Transformation 3.3.9 (Reconfiguration des connexions #3)

$$\forall t_{cnt} R_8 : \begin{cases} \textit{Ajouter } f_1 \leftarrow R_3(f) \textit{ à } P_{dst}; & \text{– Règle 3.3.4} \\ \textit{Ajouter } R_4(C) \textit{ entre } t_{mov} \textit{ et } f_1 \textit{ dans } P_{dst}; & \text{– Règle 3.3.5} \\ \textit{Ajouter } R_4(C) \textit{ entre } f_1 \textit{ et } f_2 \textit{ dans le system}; & \text{– Règle 3.3.5} \\ \textit{Supprimer } C; & \end{cases}$$

Avec t_{cnt} les threads auquels C est connectée par un port f, et f_2 les ports du process tiers auquels C est connecté.

Nettoyage Une fois ces opérations effectuées, un certain nombre de corrections doivent être faites. En effet, dans le cas des transformations 3.3.8 et 3.3.9, des connexions entre t_{mov} et des ports de P_{src} ont été supprimées. Il est donc possible que les ports de P_{src} ou P_{dst} ne soient plus connectés à l'une de leurs extrémités. Des connexions inter-process peuvent également être non-connectées. Cette situation, cependant, n'est pas automatique : il est également possible que ces process aient été initialement connectés à plusieurs connexions : dans ce cas, il restera une source et une destination aux flots associés, et le port (ainsi que la connexion associée) pourra être conservé.

Pour résoudre ce problème, nous proposons la règle 3.3.10 : on tente tout d'abord de nettoyer les connexions inter-process, puis on supprime les ports qui ne sont connectés ni en entrée, ni en sortie.

Transformation 3.3.10 (Nettoyage)

$$R_9 : \begin{cases} \forall C \in Connexions(System) \\ \quad si\ (sources(C) = \emptyset)\ ou\ (destinations(C) = \emptyset) \\ \quad supprimer\ C; \\ \forall f \in ports(P_{src}) \\ \quad si\ (source_connexions(f) = \emptyset)\ ou\ (destination_connexions(f) = \emptyset) \\ \quad supprimer\ f; \\ \forall f \in ports(P_{dst}) \\ \quad si\ (source_connexions(f) = \emptyset)\ ou\ (destination_connexions(f) = \emptyset) \\ \quad supprimer\ f; \end{cases}$$

Avec S le system contenant les process P_{src} et P_{dst}.

3.3.5 Aspect comportemental

Le comportement de l'application est affecté par l'opération *Move*, puisqu'on ajoute aux threads de P_{dst} (et qu'on retire à ceux de P_{src}) le thread t_{mov}. Il n'est pas possible de déterminer a priori le comportement des deux process : pour cela, il faut passer par une phase de calcul d'ordonnancement. Du point de vue du modèle, aucune spécification supplémentaire n'est nécessaire hormis éventuellement un nouveau calcul des priorités des threads et des ressources partagées.

Ce point est décrit plus précisément dans le chapitre 4.

3.3.6 Modifications des propriétés non-fonctionnelles

Si une connexion est créée entre deux threads (comme cela est le cas lors de l'application des règles 3.3.7 et 3.3.9), et que celle-ci passe par le system (c'est-à-dire qu'elle relie deux ports de process), il faut également l'associer à un bus. Dans notre implantation, le bus est choisi arbitrairement (précisément, c'est celui de la première connexion du system contenant les process qui sera choisi). Nous définissons donc la règle 3.3.11 ;

Transformation 3.3.11 (Association du bus) *Soit S le système contenant P_{src} et P_{dst}, soit C une connexion de S ;*
Soit c_{first} le premier élément de Connexions (S) :

R_{10} *: bus (C)* $\leftarrow bus(c_{first})$ *;*

Il est bien évident que le choix du bus à utiliser devrait être effectué de façon moins arbitraire. Pour cela, il faudrait faire une étude sur le taux d'utilisation respectif des différents bus.

3.3.7 Synthèse

Grâce aux règles de transformation définies dans cette section, nous pouvons établir un algorithme général pour appliquer l'opération Move 3.3.12.

Transformation 3.3.12 (Move : Assemblage des transformations) *Pour un move d'un thread t_{mov} du process P_{src} vers le process P_{dst}, on peut décrire l'algorithme 3.3.12. On re-*

marquera que certaines règles sont manquantes (les règles R_0, R_3 et R_4). La raison en est que celles-ci sont appelées indirectement, étant utilisées pour la définition de règles de l'algorithme.

Input: *Thread* t_{mov}, *Processes* P_{src}, P_{dst}, *System* S
Output: *Thread* t
$nom(t) \leftarrow R_1(t_{mov})$; – *Règle 3.3.2*
$subcomponents(t) \leftarrow R_2(t_{mov})$; – *Règle 3.3.3*
$ports(t) \leftarrow R_5(t_{mov})$; – *Règle 3.3.6*
foreach $C \in Connexions(t_{mov})$ **do**
  if C *est connectée a un thread de* P_{src} **then**
    $Connexions(S) \leftarrow Connexions(S) \cup R_6(C)$; – *Règle 3.3.7*
  end
  if C *est connectée a un thread de* P_{dst} **then**
    $Connexions(S) \leftarrow Connexions(S) \cup R_7(C)$; – *Règle 3.3.8*
  end
  else
    $Connexions(S) \leftarrow Connexions(S) \cup R_8(C)$; – *Règle 3.3.9*
  end
end
foreach $C \in Connexions(S)$ **do**
  $Connexions(S) \leftarrow Connexions(S) \cup R_9(C)$; – *Règle 3.3.10*
  $Connexions(S) \leftarrow Connexions(S) \cup R_{10}(C)$; – *Règle 3.3.11*
end
return t ;

3.3.8 Postconditions

Plusieurs politiques sont possibles quant à la gestion des objets (au sens général de variable) partagés avec d'autres thread du process source.

1. déplacer t_{mov}, et construire un mécanisme autorisant la synchronisation de cet objet entre P_{src} et P_{dst} ;
2. déplacer tous les threads accédant à l'objet partagé ;
3. interdire le déplacement de t_{mov}.

La solution 1 est aisément modélisable en AADL — il suffit de remonter l'objet partagé dans les sous-composants du système, et de déclarer les process source et destination comme accèdant à celui-ci — mais pose trop de problèmes de performances comparativement au modèle initial. En effet, nous devrions générer un mécanisme de type mémoire partagée, coûteux en terme de WCET et d'occupation du bus, dans un contexte où les process sont a priori répartis sur des processeurs différents. De plus, une telle solution réduit significativement l'analysabilité du problème. Si la seconde approche (solution 2) ne provoque pas de réduction générale des performances, le déplacement d'ensembles de threads réduit quant à lui la finesse de la granularité de notre approche. Nous lui préférons donc solution 3, qui interdit tout déplacement de thread partageant une donnée avec d'autres threads du même process.

3.4 Opération Merge

L'opération Merge permet de fusionner deux threads entre eux. Quand elle est appliquée à un ensemble de threads successifs, il s'agit de l'opération inverse de Split. Il n'est possible de

fusionner que selon une condition d'ordonnançabilité dite locale (la période du nouveau thread doit être supérieure à la somme des WCET des tâches — ou sous-programme terminal — qu'il effectue). Nous avons présenté cette approche dans [53].

La fusion va être effectuée en créant un automate décrivant les tâches à exécuter à chaque période, cette période étant définie comme le PGCD des périodes des threads fusionnés. L'automate décrira le comportement du thread durant une hyperpériode.

Le coût et les bénéfices de l'opération merge dépendent du système sur lequelle cette opération est appliquée. En premier lieu, toute fusion provoque la suppression d'un composant logiciel de l'intergiciel, et donc libère de la mémoire. Si les threads partageaient une connexion, celle-ci serait sérialisée et on peut donc supprimer le verrou associé - et donc réduire le WCET calculé.

Le coût induit par la gestion de l'automate est très faible, sauf dans le cas où les périodes initiales sont premières entre elle (ou proches d'être premières). Dans ce cas, on observera un grand nombre d'itérations inutiles de l'automate, pour lesquelles celui-ci n'exécutera aucune tâche. Les fusions doivent donc être choisies, entre autres, en fonction des périodes de leurs threads d'origine.

3.4.1 Préconditions

La première condition spécifie que les deux threads doivent appartenir au même process, sans quoi l'opération devrait être précédée par un $Move$ — et nous souhaitons que les opérations soient atomiques.

Par ailleurs, chaque itération du thread fusionné doit pouvoir exécuter les call sequences dont la période est un diviseur de la valeur actuelle de l'horloge. En particulier, à l'hyperpériode, toutes les call sequences périodiques seront déclenchées. En ce qui concerne les call sequences sporadiques, elles peuvent également être déclenchées dans cet intervalle. La condition de faisabilité de l'opération merge est donc :

$$\forall t_i \in T, CS = \bigcup CS_{t_i}, \sum_{cs \in CS} W_{cs} < PGCD_{cs \in CS}(W_{cs})$$

3.4.2 Règles de nommage

Construction du thread Nous définissons l'opération Merge (Me) comme la projection d'un ensemble non-vide de threads vers un thread unique : $T \subset$ Threads, Me (T) : $T \rightarrow t_{fus}$. Cette définition permet d'avancer la transformation 3.4.1, décrivant le thread créé par la fusion d'un ensemble de threads T_i.

Transformation 3.4.1 (Création du thread fusionné)

Input: *Thread Set* T
Output: *Thread* T_{fus}
$new_thread_name \leftarrow \emptyset$
foreach $T_i \in T$ **do**
| $new_thread_name \leftarrow new_thread_name \cup name(T_i)_$
end
$name(T_{fus}) \leftarrow name \cup _ < unique >$
return T_{fus}

où T est l'ensemble des threads candidats à la fusion.

Réécriture des ports En cas de collision des identifiants des ports, nous ajoutons un entier en suffixe de manière à créer un nom unique dans le système. Toutes les autres propriétés des ports restent identiques. La transformation 3.4.3 illustre la traduction de cette règle dans le modèle AADL.

Transformation 3.4.2 (Construction d'un port) R_1: $NPort \leftarrow < Pr > _k$: *kind* `port` *data_type (Pr)* ;
où :

- k *est le nombre de ports dont l'identifiant est déjà préfixé par* $< Pr >$ *dans* T_{fus}, $+1$;
- *data_type*(Pr) *est le type de la donnée transmise par* Pr ;
- *kind* : $\begin{cases} in & si\ Pr\ est\ un\ port\ en\ entrée \\ out & si\ Pr\ est\ un\ port\ en\ sortie \\ event & si\ Pr\ est\ un\ port\ d'évènements \\ data & si\ Pr\ est\ un\ port\ de\ données \end{cases}$

3.4.3 Modification des types et interfaces

Chaque port d'un thread origine doit être reproduit dans le thread final. Cette règle est illustrée par la transformation 3.4.3.

Transformation 3.4.3 (Fusion des ports)

$$R_3 : Ports(T_{fus}) \leftarrow \bigcup_{i=1..n} R_1(Ports(T_i))$$

3.4.4 Modifications de la topologie

Il existe trois types de connexions qui doivent être reconstruites :

- les connexions entre les threads fusionnés ;
- les connexions dont un thread fusionné est la source et dont un thread non-fusionné est la destination ;
- les connexions dont un thread fusionné est la destination et dont un thread non-fusionné est la source.

Pour chacune de ces connexions, la complexité de la reconfiguration dépend du fait que la call sequence contenant le sous-programme source ou destination était elle-même le résultat d'une fusion ou non. Nous différencions une call sequence issue d'une fusion par le fait qu'elle contient uniquement un wrapper (un sous-programme contenant lui-même une suite de sous-programmes), comme expliqué dans la section 3.4.5.

Le cas le plus simple est présenté dans le schéma 3.13, où deux threads candidats à la fusion, $thr1$ et $thr2$, sont connectés entre eux, et contiennent chacun un simple appel de sous-programme n'étant pas un wrapper. Dans ce cas, nous créons une connexion dont la source et la destination sont le thread fusionné. La connexion permettra de faire communiquer deux sous-programmes dans deux call sequences différentes. On remarquera que ces connexions sont autorisées par le standard, et supportées par la génération de code. Comme les sous-programmes d'origine sont encapsulés dans des wrappers, il faudra également ajouter une feature (`port` ou `parameter`) correspondant dans celui-ci. Le type et la direction de cette feature seront les mêmes que celles du port du thread auquel elles seront connectées. La figure 3.14 illustre le modèle AADL résultant de cette transformation. Dans cette figure, le sous-programme nommé *sched* représente l'ordonnanceur de mode.

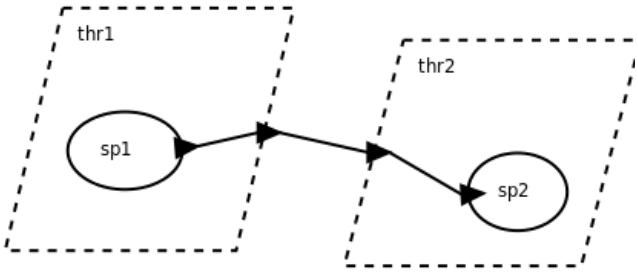


FIG. 3.13 – Connexion entre deux threads candidats à un merge

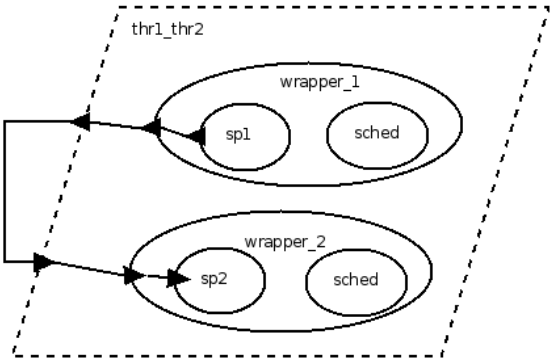


FIG. 3.14 – Auto-connexion

Si une des extrémités de la connexion est un wrapper (cf. figure 3.15), il faut retrouver tous les sous-programmes connectés au même paramètre à l'intérieur du wrapper. Le modèle résultant aurait été le même que dans l'exemple précédent, malgré les différences de modèle initial.

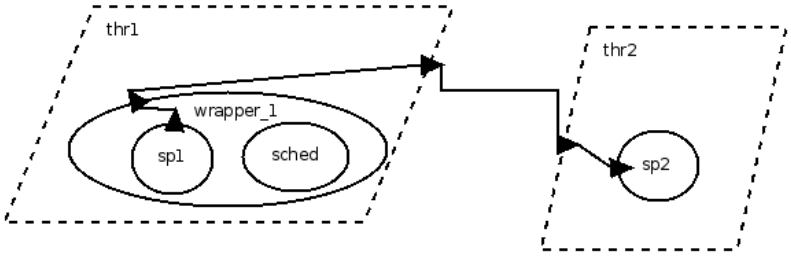


FIG. 3.15 – Connexion entre un wrapper et un sous-programme unique

Le schéma 3.16 illustre un exemple de connexion vers des threads non fusionnés. Dans cette figure le thread source $thr1$ va être fusionné avec le thread $thr2$, mais pas avec le thread destination $thr3$. Dans ce cas, les connexions locales à $thr3$ ne sont pas modifiées, mais on construit dans $thr1$ le port correspondant sur le wrapper, comme montré en 3.17. Un exemple un peu plus complexe est donné dans le schéma 3.18. Cette fois, la connexion locale dans $thr1$ accède à un port lui-même connecté à plusieurs threads, dont l'un est fusionné et l'autre non. Dans ce cas, il ne faut générer qu'un seul port dans le wrapper qui contiendra le sous-programme $sp1$ de $thr1$, et une seule connexion de ce port vers le port du thread fusionné associé. La multiplicité de la connexion se retranscrit au niveau du process, et non du thread, comme le montre le schéma 3.19.

Nous résumons ces opérations par la transformation 3.4.11, applicable à chaque connexion CN_{CS_i} dont une des extrémités est connectée à un sous-programme SP_j d'une connexion fusionnée CS_i. On définit par P_T le port de T auquel est connecté CN_{CS_i}.

Transformation 3.4.4 (Reconstruction des connexions)

R_2: { *Si la feature P_j de SP_j n'est pas accédée par une autre connexion antérieure ajouter une feature P à W_i le wrapper associé à CS_i.*
Sinon $< P >$ est défini comme étant la feature créée pour la connexion antérieure ;
Construire une connexion de P_j vers P;
Construire une connexion de P vers le port de T_{fus} correspondant à P_T.

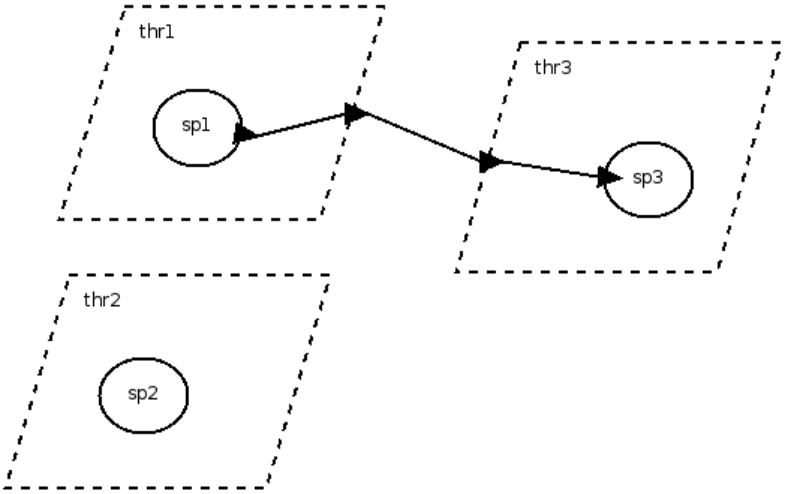


FIG. 3.16 – Connexion entre un thread à fusionner et un thread non fusionné

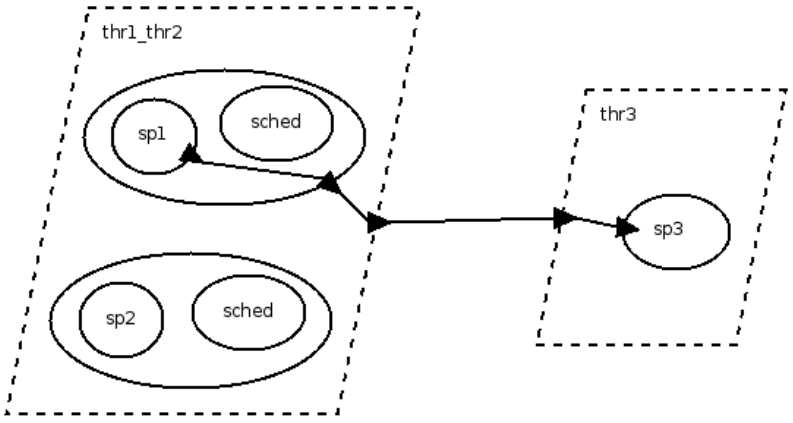


FIG. 3.17 – Connexion entre un thread à fusionner et un thread non fusionné (résultat)

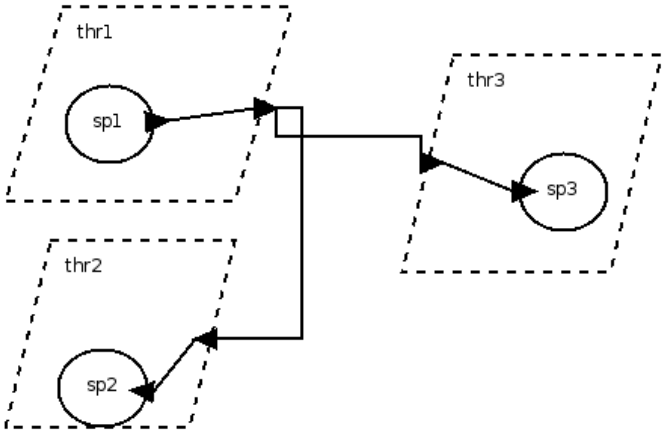


FIG. 3.18 – Connexions multiples

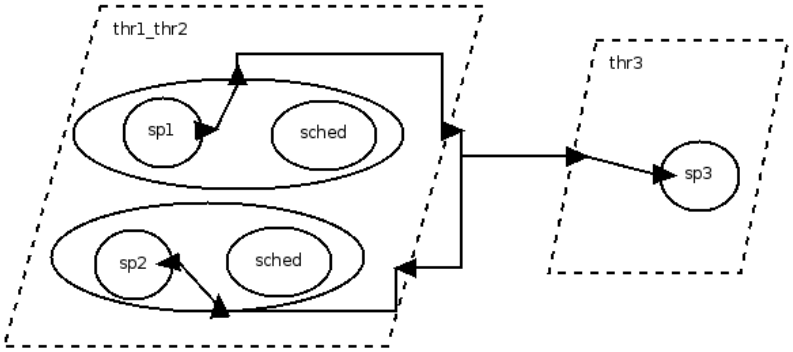


FIG. 3.19 – Connexions multiples (résultat)

3.4.5 Aspect comportemental

Fusion des call sequences Nous avons vu que notre automate doit exécuter à chaque itération un ensemble variable de sous-programmes. Pour mener ces exécutions à bien, il faut construire dans le thread fusionné un ensemble de `call sequences` qui est en bijection avec l'union des `call sequences` des threads d'origine. Nous en tirons donc la règle comportementale 3.4.1.

Comportement 3.4.1 (Fusion des call sequences)

$$\forall cs_i \in CS_T R_4 : CS_{T_{fus}} \leftarrow cs_i$$

où T est l'ensemble des threads fusionnés.

Dans le cas où la call sequence contient plusieurs appels, nous encapsulons cet appel (et la surcharge de code induite par notre mécanisme) dans un sous-programme nommé *wrapper*, dont la construction est détaillée plus loin. Dans le cas où une call sequence ne contient qu'un seul appel à un sous-programme, nous distinguons deux cas :

- si le sous-programme est un *wrapper*, nous récupérons ses appels, les encapsulons dans un nouveau wrapper, et y ajoutons la nouvelle surcharge induite par le code ;

– dans le cas contraire, nous encapsulons cet unique appel, en y ajoutant la nouvelle surcharge induite par le code.

Comportement 3.4.2 (Construction d'un wrapper)

$$C_5: \begin{cases} \textit{Si } SP_{CS} \textit{ est un wrapper (unique)} \\ \quad \textit{Si } CS_i \textit{ provient d'une fusion antérieure} \\ \quad \textit{On copie dans CS' tous les éléments de la call sequence de } SP_{CS}\textit{, sauf le dernier;} \\ \quad \textit{Sinon On copie dans CS' tous les éléments de la call sequence de } SP_{CS}\textit{;} \\ \textit{Sinon On copie dans CS' tous les éléments de } SP_{CS}\textit{.} \\ \textit{On ajoute un appel au sous-programme } OH\ (CS_i) \\ \textit{On construit un wrapper contenant CS'} \end{cases}$$

où CS est la call sequence considérée, et SP_{CS} les sous-programmes appelés par CS.

La règle 3.4.2 présente les différents cas de construction de wrapper pour chaque call sequence CS. On notera que si la call sequence copiée provient d'un thread déjà fusionné, elle aura comme dernier élément un appel de sous-programme particulier, qui ne correspond pas à un élément de l'architecture initiale. Cet élément, ici noté $OH(CS_i)$, est un appel permettant de maintenir le comportement du thread. Nous fournissons ci-dessous tous les détails sur cet élément.

Déclenchement des call sequences Chacune des calls sequence du thread fusionné doit être déclenchée lors de la réception d'un évènement qui lui est propre. Le standard AADL supporte la définition de plusieurs call sequences sporadiques (dans des threads sporadiques ou hybrides), entre autres méthodes en utilisant la propriété `compute_entrypoint_call_sequence`, qui associe un `event port` (ou `event data port`) entrant et une call sequence. L'exemple 3.1 déclare que la call sequence *cs* est déclenchée sur réception d'un évènement par le port *in_msg*. Nous pouvons donc avancer la transformation 3.4.5 pour chaque call sequence CS.

```
thread T
 features
   in_msg : in event port;
end T;

thread implementation T.i
calls
  cs : {main : subprogram wrapper.i;}
properties
  compute_entrypoint_call_sequence => reference ( cs )
      applies to in_msg;
end T.i;
```

Listing 3.1 – Associer un port et une call sequence avec AADL

Transformation 3.4.5 (Déclenchement des call sequences sporadiques)

$$R_4: \begin{cases} \textit{si CS est sporadique} \\ \forall p_j \in I_{CS} \\ \quad \textit{associer } p_j \textit{ et CS avec une propriété compute_entrypoint_call_sequence} \end{cases}$$

Dans le cas de call sequences périodiques, comme spécifié dans le standard AADL, l'exécutif sous-jacent va exécuter la call sequence qui n'est pas liée à un port de type `event`. Pour un

`mode` donné, il n'est possible d'avoir qu'une seule call sequence correspondant à cette description. Comme les merges successifs invalident cette hypothèse, nous proposons de construire de nouveaux modes pour alterner l'exécution des call sequences périodiques. L'exemple 3.2 illustre comment la propriété `compute_entrypoint_call_sequence` permet d'associer une call sequence à un mode. La transformation 3.4.6 résume les opérations à effectuer dans le cas de call sequences périodiques. On notera cependant que nous faisons l'hypothèse qu'aucun mode n'existe initialement dans le thread. Il y a en fait deux cas à distinguer :

- s'il existe des modes dans le thread d'origine, mais que ces modes ont été créés par merge successifs, nous ne perdons pas d'information, puisque le nouvel ordonnancement prendra en compte les fusions précédentes ;
- s'il existe des modes dans le thread d'origine, mais que ces modes n'ont pas été créés par merge successifs, alors nous perdons l'information liée à ces modes.

Le second point est une limitation de l'algorithme actuel, qui peut être assez simplement amélioré en faisant le produit cartésien des nouveaux et des anciens modes.

```
thread T
end T;

thread implementation T.i
calls
  cs_1 : {main1 : subprogram wrapper.i1;}
  cs_2 : {main2 : subprogram wrapper.i2;}
modes
    mode_1 : initial mode;
    mode_2 : mode;
properties
  compute_entrypoint_call_sequence => reference ( cs_1 )
      in (mode_1);
  compute_entrypoint_call_sequence => reference ( cs_2 )
      in (mode_2);

end T.i;
```

Listing 3.2 – Associer un mode et une call sequence avec AADL

Transformation 3.4.6 (Déclenchement des call sequences périodiques)

$$\forall cs_i \in CS_T R_5 : \begin{cases} On\ ajoute\ à\ T_{fus}\ un\ mode\ unique\ M_i \\ On\ ajoute\ à\ T_{fus}\ une\ propriété\ p_{trigger} = compute_entrypoint_call_sequence \\ On\ associe\ p_{trigger}\ à\ M_i \end{cases}$$

où T est l'ensemble des threads fusionnés.

Gestion de l'automate Un sous-programme (*ordonnanceur de mode*) appelé à la fin de toutes les call sequences périodiques se charge de calculer le mode suivant dans le même quantum (intervalle de temps égale à la période du thread final, où chaque tâche s'exécute au plus une fois). Cet appel à un sous-programme définit la surcharge propre au code, comme précisé dans la transformation 3.4.7. Un autre sous-programme est ajouté pour indiquer que l'on passe au quantum suivant. Pour cela, il nous faut également ajouter une call sequence spéciale et le mode correspondant. Le code AADL 3.3 illustre la modélisation de l'ordonnanceur de mode, ainsi que son utilisation après l'appel au code fonctionnel d'un wrapper (défini pour chaque call sequence CS_i par la règle 3.4.3), et l'impact sur le modèle AADL est présenté dans la transformation 3.4.8. Nous voyons qu'il fait appel à une fonction issue d'un package Ada. Cette fonction est générée par notre algorithme d'ordonnancement, et est présentée plus loin.

Transformation 3.4.7 (Ordonnanceur de mode)

$$R_7 : \begin{cases} \textit{Ajouter un sous-programme } mode_scheduler; \\ \textit{Ajouter un sous-programme } next_period. \end{cases}$$

```
subprogram mode_scheduler
end mode_scheduler;

subprogram next_period
end next_period;

subprogram implementation mode_scheduler.i
  properties
    source_language => ada95;
    transformations::source_name => "software_thr_mode.mode_scheduler";
end mode_scheduler.i;

subprogram implementation next_period.i
  properties
    source_language => ada95;
    transformations::source_name => "software_thr_mode.next_period";
end next_period.i;

subprogram implementation wrapper.i
  calls
    CS1 :
    {sp1       : subprogram Any_Job;
     sched     : subprogram mode_scheduler.i;}
;
end wrapper.i;
```

Listing 3.3 – Ordonnanceur de mode : partie AADL

Comportement 3.4.3 (Appel à l'ordonnanceur de mode)

$$C_6 : OH(CS_i) \leftarrow sched$$

Transformation 3.4.8 (Appels du wrapper #2)

$$\forall CS_i \in CS_T R_8 : Calls(W_i) \leftarrow Calls(CS_i) \cup sched$$

où T est l'ensemble des threads fusionnés

Le code 3.4 illustre les modalités de passage au quantum suivant. La règle 3.4.9 décrit la modification à apporter au modèle fusionné.

```
thread implementation T.i
calls
  cs_1 :
    {main_1 : subprogram wrapper_1.i;} in modes (mode_1);
  cs_2 :
    {main_2 : subprogram wrapper_2.i;} in modes (mode_2);
  jump :
    {next_period : subprogram next_period.i;} in modes (end_of_quantum);
modes
  mode_1          : initial mode;
  mode_2          : mode;
  end_of_quantum  : mode;
properties
  compute_entrypoint_call_sequence => reference ( cs_1 ) in (mode_1);
```

```
  compute_entrypoint_call_sequence => reference ( cs_2 ) in (mode_2);
  compute_entrypoint_call_sequence => reference ( jump ) in (end_of_quantum);
end T.i;
```

Listing 3.4 – Passage au quantum suivant

Comportement 3.4.4 (Passage au quantum suivant : impact comportemental)

$$C_7 : \begin{cases} CS_{next_period} := (\emptyset, \emptyset, next_period) \\ CS_{T_{fus}} \leftarrow CS_{T_{fus}} \cup CS_{next_period} \end{cases}$$

Transformation 3.4.9 (Passage au quantum suivant)

$$R_9 : \begin{cases} Ajouter\ un\ mode\ M := end_of_quantum\ à\ T_{fus}; \\ Ajouter\ une\ call\ sequence\ jump\ dans\ T_{fus}; \\ Ajouter\ une\ propriété\ compute_entrypoint_call_sequence\ qui\ associe\ jump\ à\ end_of_quantum; \end{cases}$$

Génération des tâches périodiques Nous avons vu que la surcharge de code pour chaque call sequence correspond à l'appel à l'ordonnanceur de mode. Par ailleurs, un appel à la fonction *next_period* est effectué à la fin de chaque quantum. Il importe donc que la complexité de ces fonctions soit en $O(1)$.

Avec le générateur de code Ocarina, le code généré par le modèle 3.4 comportera une instruction `case` vérifiant le mode courant, et exécutant le code correspondant, illustré par le code 3.5, où $Software_wrapper_1_i$ et $Software_wrapper_2_i$ sont les wrappers contenant le code à exécuter dans leurs modes respectifs.

```
case Current_Mode is
  when mode_1 =>
    Software_wrapper_1_i;

  when mode_2 =>
    Software_wrapper_2_i;

  when stop_mode =>
    Software_Next_Period;

end case;
```

Listing 3.5 – Code-Source Ada généré par Ocarina

Dans notre version, nous ajoutons une boucle autour du `case`, puisque nous pouvons exécuter plusieurs call sequences durant un même quantum. La sortie de cette boucle est contrôlée par une variable mise à jour par la fonction générée *next_period*. Chaque appel à un wrapper se termine (implicitement) par un appel à *mode_scheduler*. Une illustration de notre solution est présentée dans le code 3.6, où la variable *R_Continue* (variable indiquant l'état de l'ordonnanceur, déclarée dans un package généré par le processus de fusion) est mise à *false* par effet de bord par l'appel à *Software_Next_Period*.

```
R_Continue := True;
loop
  case Current_Mode is
    when mode_1 =>
      Software_wrapper_1_i;

    when mode_2 =>
```

```
      Software_wrapper_2_i;

    when stop_mode =>
      Software_Next_Period;

  end case;
  exit when not R_Continue;
end loop;
```

Listing 3.6 – Code-Source Ada correct

Pour résoudre le problème de la complexité des fonctions *next_period* et *mode_scheduler*, nous établissons un ordonnancement des différentes call sequences à effectuer à chaque quantum sur une hyperpériode. Pour établir notre ordonnancement, nous avons besoin d'une relation d'ordre entre les call sequences s'exécutant durant un même quantum. Pour cela, nous utilisons leur priorité originale. Nous insérons séquentiellement le résultat de cet ordonnancement dans un tableau de mode (en ajoutant *stop_mode* à la fin de chaque quantum), que nous générons dans un package Ada séparé de l'application. C'est également dans ce package que seront définies les fonctions *next_period* et *mode_scheduler* référencées par le modèle AADL. Le code-source de ces fonctions est invariant et est donné en 3.7, où *change_mode* se contente de mettre à jour la variable *current_mode*. Elles sont définies comme suit :

- **mode_scheduler** est un simple incrémenteur sur le tableau des modes ;
- **next_period**
  - met à faux la variable indiquant qu'il reste des call sequences à exécuter ;
  - réinitialise l'itérateur.

La complexité de ces méthodes est évidement en $O(1)$, puisqu'il s'agit dans le pire des cas d'une comparaison, d'une addition, de deux affectations et d'un accès direct à un tableau.

```
Current_Iteration : Integer := 0;

procedure Mode_Scheduler is
 begin
    Next_Iteration;
    Change_Mode (Schedule_Table (Current_Iteration));
 end Mode_Scheduler;

 procedure Next_Iteration is
 begin
    if Current_Iteration < Array_Size then
       Current_Iteration := Current_Iteration + 1;
    else
       Current_Iteration := 0;
    end if;
 end Next_Iteration;

 procedure Next_Period is
 begin
    R_Continue := False;
 end Next_Period;
```

Listing 3.7 – Code-Source de mode_scheduler et next_period

Prise en compte des call sequences sporadiques dans le code généré Dans le cas de call sequence sporadiques (dans un thread hybride par exemple), le code généré sera de la forme indiquée dans le listing 3.9, correspondant au modèle AADL 3.8. Dans cet exemple, la variable `Port` contient le premier évènement en attente de traitement. Pour unifier le traitement des call sequences sporadiques et périodiques, ces dernières sont considérées comme sporadiques, et

déclenchées sur un évènement spécial dit *Period_Event*, envoyé par la runtime à chaque période du thread. Dans les autres cas, les noms des évènements sont les mêmes que les noms des ports dont ils proviennent (dans l'exemple, *msg_in*). Comme la call sequence associée à ce dernier ne dépend pas d'un mode particulier, elle se déclenchera quelle que soit la valeur de *Current_Mode*.

```
thread T
features
  msg_in : in event port;
end T;

thread implementation T.i
calls
  cs_1 :
    {main_1 : subprogram wrapper_1.i;}
      in modes (mode_1);
  cs_2 :
    {main_2 : subprogram wrapper_2.i;}
      in modes (mode_2);
  jump :
    {next_period : subprogram next_period.i;}
      in modes (stop_mode);
  cs_3 :
    {main_3 : subprogram wrapper_3.i;};

modes
  mode_1    : initial mode;
  mode_2    : mode;
  stop_mode : mode;
properties
  compute_entrypoint_call_sequence => reference ( cs_1 )
      in (mode_1);
  compute_entrypoint_call_sequence => reference ( cs_2 )
      in (mode_2);
  compute_entrypoint_call_sequence => reference ( jump )
      in (stop_mode);
  compute_entrypoint_call_sequence => reference ( cs_3 )
      applies to ( msg_in );
end T.i;
```

Listing 3.8 – Thread AADL hybride

```
case Port is

  when Period_Event =>
    R_Continue := True;
    loop
      case Current_Mode is
        when mode_1 =>
          Software_wrapper_1_i;

        when mode_2 =>
          Software_wrapper_2_i;

        when stop_mode =>
          Software_Next_Period;

      end case;
    exit when not R_Continue;
    end loop;

  when Msg_In =>
    case Current_Mode =>
      when others =>
       Software_wrapper_3_i;
```

```
    end case;

end case;
```

Listing 3.9 – Code-Source d'un thread hybride

Notre contribution à ce mécanisme est l'extension du mécanisme préexistant pour la propriété standard AADL `compute_entrypoint` à la propriété `compute_entrypoint_call_sequence`. Le mécanisme original est décrit précisément par son auteur en [91].

3.4.6 Propriétés non fonctionnelles

Gestion de l'automate Nous avons expliqué comment nous définissions les différentes call sequences à exécuter, et comment nous les associons à des modes ou à des ports d'évènements rentrants. Dans le premier cas, cependant, il nous reste à définir la façon dont nous commandons l'alternance des modes. Pour cela, nous divisons la durée maximale d'exécution unique (*hyperpériode*) en segments minimum (*quantum*) durant lesquels l'exécution d'un sous-ensemble des call sequences du thread fusionné peut être déclenchée. La partie périodique du comportement du thread fusionné peut être défini totalement avec son hyperpériode. Nous en tirons la règle 3.4.5, ainsi que la transformation 3.4.10.

Comportement 3.4.5 (Période)

$$C_1 : P_T \leftarrow PPCM(P_{CS_i})$$

Transformation 3.4.10 (Assignation de la période du thread)

$$R_6 : Period(T_{fus}) \leftarrow PPCM(P_{CS_T})$$

Gestion des priorités Dans un thread typique contenant une call sequence, la call sequence s'exécute à la priorité du thread qui l'héberge. Dans un thread résultant de la fusion de plusieurs threads, donc contenant plusieurs priorités, nous devons définir la priorité du thread. Nous avons choisi de ne pas recourir à des changements dynamiques de priorité, pour préserver le déterminisme du système et suivre ainsi les recommandations du profile Ravenscar [44]. Pour cela, nous définissons une priorité statique pour le thread.

Parmi les valeurs potentielles, nous avons choisi la solution suivante : le thread hérite de la plus basse priorité parmi les threads des call sequences fusionnées. Cependant, en vue de conserver le comportement du système, nous procédons à une augmentation de priorité en utilisant un verrou dédié, configuré avec le Priority Ceiling Protocol, que nous nous nommons $priority_shifter$. Lors de l'aquisition du verrou, la priorité du thread deviendra celle du verrou. Pour chaque priorité unique P_i différente de celle du thread final et possédée par une call sequence i fusionnée, nous créons une variable locale $priority_shifter_i$ dont la priorité est P_i. Cette variable est saisie au début de l'exécution des call sequences de même priorité, et relachée à la fin. Les règles 3.4.6 et 3.4.7 définissent les aspects formels de cette opération, et les transformations 3.4.11 et 3.4.12 son impact sur le modèle AADL. Les conséquences de l'introduction de ce protocole est détaillé dans la section 3.4.8.

Le diagramme de Gantt 3.20 illustre le déroulement des threads munis de $priority_shifter$. Dans ce scénario, les threads $thr2$ et $thr3$ sont non-fusionnés, et le thread $thr1$ l'est (et contient

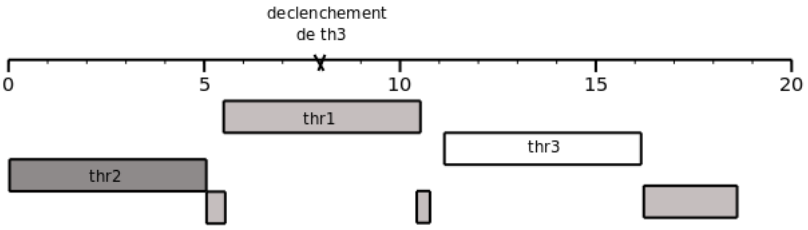


FIG. 3.20 – Utilisation d'un priority_shifter : diagramme de Gantt

deux call sequences). Les threads $thr1$ et $thr2$ sont tous deux déclenchés à $t = 0$, tandis que $thr3$ est déclenché à $t = 7$. Nous avons les relations suivantes pour les priorités :

$$Min(priorit(thr1)) < priorit(thr2) < priorit(thr3)$$

et

$$Max(priorit(thr1)) > priorit(thr3) > priorit(thr2)$$

Nous voyons que le premier thread à prendre la main est $thr2$, car à l'initialisation la priorité de $thr1$ est la priorité minimale de ses call sequences. Quand il finit sa call sequence à $t = 5$, $thr1$ prend la main, et exécute sa première call sequence. Comme celle-ci a une priorité effective supérieure à la priorité de $thr2$, elle est associée à un $priority_schifter$. Durant son exécution, $thr1$ change donc de priorité. Il n'est donc pas préempté par $thr3$ à $t = 7$. Quand il a fini sa première call sequence, il relâche le $priority_schifter$, et reprend sa priorité initiale. $thr3$ reprend donc la main, et $thr1$ devra attendre la fin de l'exécution de la call sequence de ce dernier pour exécuter sa dernière call sequence.

Comportement 3.4.6 (Priorité)

$$C_2 : P \leftarrow Min(P_i)$$

où P_i sont les priorités des call sequences des threads fusionnés.

Comportement 3.4.7 (Changement de priorité) *Si la priorité d'une call sequence CS est supérieure à la priorité du thread final, ajouter un objet priority_ shifter au modèle AADL, configuré avec PCP et dont la priorité est P_{CS}. Le thread exécute la séquence suivante :*

$$C_3 : CS \rightarrow (PS_{CS}.get(), CS, PS_{CS}.release())$$

Transformation 3.4.11 (Priorité)

$$R_{10} : Priorit(T_{fus}) \leftarrow Min(Priorit(T))$$

où T est l'ensemble des threads fusionnés.

Transformation 3.4.12 (Changement de priorité)

$$R_{11} : \begin{cases} \text{S'il n'existe pas de type de donnée de priorité } P_{CS_i}\text{, créer une donnée } D; \\ \text{Ajouter une instance } D_i \text{ de } < D > \text{ dans } T_{fus}; \\ \text{Ajouter la séquence d'appel } CS_i \text{ dans } < D >; \\ \text{Replacer la call sequence } CS_i \text{ dans } T_{fus} \text{ par une nouvelle} \\ \text{call sequence appelant la call sequence correspondant de } D_i. \end{cases}$$

3.4.7 Synthèse

Grâce aux règles de transformation définies dans cette section, nous pouvons établir un algorithme général pour appliquer l'opération Merge 3.4.13.

Transformation 3.4.13 (Assemblage des transformations) *Pour un merge de deux threads, on peut décrire l'algorithme suivant :*

Input: *Threads* T_1, T_2
Output: *Thread* T_3, *merge de* T_1, T_2
$< T_3 > \leftarrow < T_1 > _ < T_2 >$; – *Règle 3.4.1*
$CS \leftarrow R_3(CS_{T_1}) \bigcup R_3(CS_{T_2})$; – *Règle 3.4.3*
$T_3 \leftarrow R_8(T_3)$; – *Règle 3.4.8*
$T_3 \leftarrow R_9(T_3)$; – *Règle 3.4.9*
foreach $CS_i \in CS$ **do**
  $CS_i \leftarrow R_7(CS_i)$; – *Règle 3.4.7*
  $CS_i \leftarrow R_2(CS_i)$; – *Règle 3.4.4*
  if $P_{CS_i} > P$ **then**
    $CS_i \leftarrow R_{10}(CS_i)$; – *Règle 3.4.11*
    $CS_i \leftarrow R_{11}(CS_i)$; – *Règle 3.4.12*
  end
end
foreach $CS_i \in CS | CS_i periodic$ **do**
  $CS_i \leftarrow R_5(CS_i)$; – *Règle 3.4.6*
end
foreach $CS_i \in CS | CS_i sporadic$ **do**
  $CS_i \leftarrow R_4(CS_i)$; – *Règle 3.4.5*
end
$T3 \leftarrow R_6(T_3)$ – *Règle 3.4.10*
return T_3 ;

Nous pouvons également exprimer les modifications purement comportementales de l'application grâce à l'algorithme 3.4.8.

Comportement 3.4.8 (Assemblage des règles de comportements) *Pour un merge de*

deux threads, on peut décrire l'algorithme suivant :

Input: *Threads* T_1, T_2
Output: *Thread* T_3, *merge de* T_1, T_2
$period(T_3) \leftarrow C_1(T_1, T_2)$; – *Règle 3.4.5*
$priority(T_3) \leftarrow C_2(T_1, T_2)$; – *Règle 3.4.6*
foreach $CS \in Call_Sequences(T_1 \cup T_2)$ **do**
  $Call_Sequences(T_3) \leftarrow Call_Sequences(T_3) \cup C_4(CS)$; – *Règle 3.4.1*
end
foreach $CS \in Call_Sequences(T_3)$ **do**
  $CS \leftarrow C_5(CS)$; – *Règle 3.4.2*
  $CS \leftarrow C_6(CS)$; – *Règle 3.4.3*
  $CS \leftarrow C_7(CS)$; – *Règle 3.4.4*
  if *Priority (CS) > min (Priority* (T_3)) **then**
    $CS \leftarrow C_3(CS)$; – *Règle 3.4.7*
  end
end
return T_3 ;

Contraintes et limitations Comme nous l'avons vu tout au long de cette section, l'approche que nous proposons présente un certain nombre de limitations.

En premier lieu, il faut vérifier la condition $\sum_{cs_i \in CS} WCET(cs_i) < I_{T_{fus}}$ pour pouvoir trouver un ordonnancement des call sequences. Comme $I_{T_{fus}}$ est égale au quantum, et donc au plus grand commun diviseur des périodes des call sequences, cela implique qu'on ne peut raisonnablement pas fusionner des threads dont les périodes respectives sont premières entre elles. C'est un aspect dont l'évaluation des opérations devra prendre en compte (cf. chapitre 4).

Nous avons également vu que, dans la description du modèle initial, nous exploitions la propriété standard `compute_entrypoint_call_sequence`. Il existe cependant d'autres manières d'associer l'exécution de code exécutif (pas forcément de call sequences) conformes au standard AADL. Ces méthodes sont :

- utiliser la propriété `compute_entrypoint` qui prend un chaine de caractère en paramètre, et lui indiquer directement quelle procédure exécuter dans le code-source, qui est associée à un port ou à un mode ;
- utiliser la propriété `compute_entrypoint_subprogram` qui prend un sous-programme AADL en paramètre, et lui associe un port et/ou un mode ;
- spécifier une unique call sequence périodique, qui sera exécutée implicitement.

Nous supportons la dernière méthode, mais pas les deux premières. Tous les modèles formellement justes (au sens du formalisme AADL) ne peuvent donc pas être optimisés avec notre algorithme. Il est possible cependant, au prix d'un travail raisonnable, d'étendre le support aux deux premières méthodes. Dans le cas de la première méthode, cependant, nous pensons qu'il s'agit d'une mauvaise pratique, puisqu'il est impossible d'associer des propriétés à la procédure exécutée.

Nous supposons également que la profondeur des connexions est au maximum de 1. Typiquement, cela signifie qu'un composant contiendra les connexions entre ses sous-composants, mais pas les connexions entre leurs propres composants. Toutes les topologies peuvent êtres décrites avec cette approche. Nous pensons que spécifier des connexions de profondeur supérieure à 1 est une mauvaise pratique, car elle fait des hypothèses sur l'implantation des sous-composants.

Une autre limitation, plus restrictive, concerne l'utilisation des modes. En l'état, les modes

pré-déclarés sont perdus après fusion. Il s'agit d'une limite importante de notre implantation, qui devrait être levée dans le cadre d'un livrable complet.

Concernant l'implantation du code (Ada ou AADL) généré, d'autres points pourraient être améliorés. En premier lieu, le critère de tri des séquences d'appels utilisé par l'ordonnanceur n'est clairement pas optimal : en effet il ne prend pas en compte les connexions, causant ainsi un risque de retard systématique des traitements (ou traitement sur des données obsolètes). Pour corriger ce point, un tri topologique devrait être effectué sur les call sequences, permettant de désigner une relation d'ordre partiel entre elles.

Une autre amélioration possible serait de construire une call sequence unique pour chaque combinaison de call sequences effectivement appelées durant l'hyperpériode. On pourrait ainsi réduire à 1 le nombre maximum d'appels à l'ordonnanceur de mode à chaque quantum.

Finalement, l'utilisation de PCP pour simuler les changement de priorité ne produit pas un comportement conforme au comportement initial. En effet, si un thread fusionné $T1$ de priorité $P1$, appelant une méthode de priorité $P3$ est en concurrence avec un autre thread $T2$ de priorité $P2$, avec $P1 < P2 < P3$, c'est $T2$ qui prendra la main, et non $T1$ comme dans le système non fusionné. En effet, le mutex faisant passer le thread fusionné à $P3$ est saisi après que le thread $T1$ ait commencé son exécution - avant cela, sa priorité effective est celle du thread, donc $P1$. Cete approche cependant est utile pour empècher la call sequence fusionnée de priorité théorique $P3$ de se faire elle-même préempter, si elle est en cours d'exécution. Passer à un ordonnancement non préemptif permettrait d'obtenir le même résultat, et ce, sans ajouter de mutex (facilitant ainsi l'analyse).

3.4.8 Postconditions

Nous avons vu que les transformations que nous avons définies avaient un impact sur le comportement de l'application. En premier lieu, le thread fusionné hérite de la priorité minimale parmi les tâches qu'il contient. Si la priorité de ces tâches change ensuite une fois qu'elles ont pris la main, cette priorité minimale est celle des tâches du thread au moment du déclenchement. Une tâche peut donc être préemptée par n'importe quelle tâche de plus haute priorité *que la priorité minimale de son thread*. La règle (1) résume ces considérations.

Cette règle, cependant, ne s'applique qu'aux threads périodiques, car les tâches sporadiques d'un thread sont toujours exécutées après tous les threads périodiques, comme illustré dans le listing 3.9. Les tâches sporadiques des autres threads peuvent cependant préempter un thread périodique, si celles-ci sont supérieures à sa priorité effective, comme montré dans la règle (2).

Une tâche sporadique peut être préemptée par une tâche périodique de priorité supérieure à sa priorité effective, comme dans le cas de tâches périodiques. La règle (1) est donc toujours valable dans ce cas.

Une tâche sporadique, peut également être préemptée (avant son déclenchement) par toute tâche sporadique du même thread, quelle que soit sa priorité, pour peu que celui-ci soit déclenché juste avant le premier. Il peut également être préempté par toute tâche de priorité supérieure, que ce soit dans le même thread ou dans un autre. Il interrompra cependant les tâches de priorité inférieure dans les autres threads. Les règles (3) et (4) résument ces aspects.

$$\begin{cases} (1) & C1 & \sum_{t_j \in hp_{min_{thread(t_i)}}(Periodic(T))} WCET(t_j) \\ (2) & C2 & \sum_{t_j \in hp_{min_{thread(t_i)}}(Sporadic(T)) \wedge thread(t_j)=thread(t_i)} WCET(t_j) \\ (3) & C3 & \sum_{t_j \in lp_{min_{thread(t_i)}}(Sporadic(T)) \wedge thread(t_j)=thread(t_i)} WCET(t_j) \\ (4) & C4 & \sum_{t_j \in hp_{min_{thread(t_i)}}(Sporadic(T)) \wedge t_j \neq t_i} WCET(t_j) \end{cases}$$

où :

- $hp_n(T)$ retourne l'ensemble des tâches de priorité supérieure à n dans l'ensemble de tâches T ;
- $lp_n(T)$ retourne l'ensemble des tâches de priorité inférieure à n dans l'ensemble de tâches T ;
- min_T retourne la priorité minimale dans les tâches de l'ensemble T ;
- $thread(t_i)$ retourne l'ensemble des tâches (donc le thread) auquel appartient la tâche t_i.

Finalement, nous pouvons résumer les différents temps de réponse par les règles suivantes :

$$RT(t_i) < \begin{cases} C1 + C2 & \text{si } t_i \text{ est périodique ;} \\ C1 + C3 + C4 & \text{si } t_i \text{ est sporadique.} \end{cases}$$

3.5 Autres opérations

3.5.1 Opération Split

L'opération split peut être effectuée sur des threads contenant plus d'une tâche. Elle créera le même nombre de nouveaux threads, chacun exécutant une seule tâche. La période des nouveaux threads sera égale à la période spécifiée pour leur tâche, et non à celle du thread dont ils sont issus.

Cette opération permet d'étendre l'ensemble des systèmes ordonnançables, car nous autorisons de nouvelles préemptions. Cependant, puisque nous allouons $n - 1$ threads, où n est le nombre de tâches exécutées par le thread d'origine, nous augmentons l'empreinte mémoire proportionnellement.

De plus, si des communications existent entre les tâches, celles-ci doivent être transformées en connexions, et donc nécessitent l'ajout de mutex, réduisant ainsi l'analysibilité du système et rendant plus pessimiste l'évaluation du WCET.

Les figures 3.21 et 3.21 illustrent l'effet de l'opération *Split* sur un thread contenant deux call sequences (toutes deux appelant un wrapper). On remarquera que, étant en présence de ports de données pour ces deux call sequences, nous pouvons supprimer les ports non utilisés dans les deux threads résultants de l'opération.

Dans le cadre de cette thèse, nous n'avons pas implanté l'opération `Split`, faisant l'hypothèse que les modèles initiaux ne comportaient qu'une seule call sequence par thread. Cette hypothèse empêche certains usage du processus d'optimisation, puisqu'il interdit de faire les retours en arrière dans l'espace des solutions visitées. Dans le cadre d'une optimisation en une passe, cela n'est pas gênant, mais il existe d'autres cas d'utilisations réalistes : en particulier, en cas de changement de plateforme matérielle d'une application existante (et optimisée avec notre processus), les espaces déjà visités pourraient être perdus, et la solution optimale pourrait s'y trouver. Implanter cette procédure sera donc une tâche nécessaire pour l'avenir.

Cependant, en l'état actuel du standard AADL, nous ne pouvons pas retrouver tous les flots passant par un thread. En effet, dans le cas où il est muni de ports d'évènements (`event` ou `data event`), certaines connexions effectives peuvent ne pas apparaitre dans le modèle. C'est

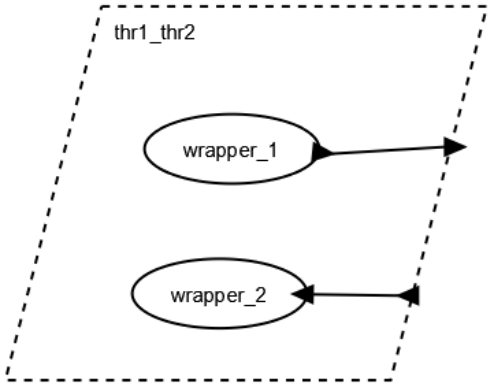


FIG. 3.21 – Thread contenant 2 call sequences

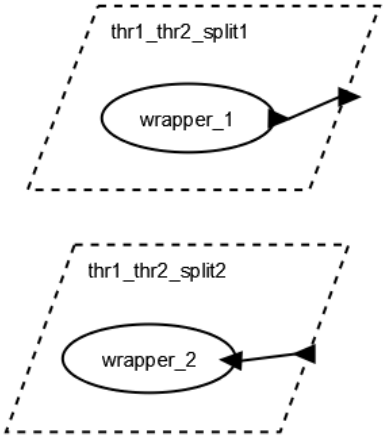


FIG. 3.22 – Split d'un thread contenant 2 call sequences

le cas quand le code-métier utilise les primitives d'envoi d'évènements. Cette situation donne lieu à des modèles suivant la forme illustrée par la figure 3.23. Dans ce cas particulier, faire une séparation du thread impliquerait de dupliquer de façon potentiellement inutile un grand nombre de ports dans les threads (la figure 3.24 en est une illustration), et induirait par là une surcharge significative (allocation des tampons liés au port, interrogation à chaque nouvelle exécution du thread...).

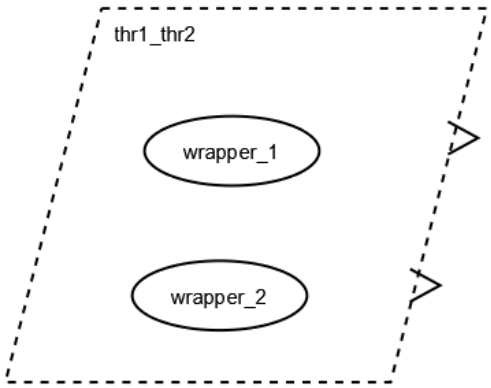


FIG. 3.23 – Thread contenant 2 call sequences et des ports events

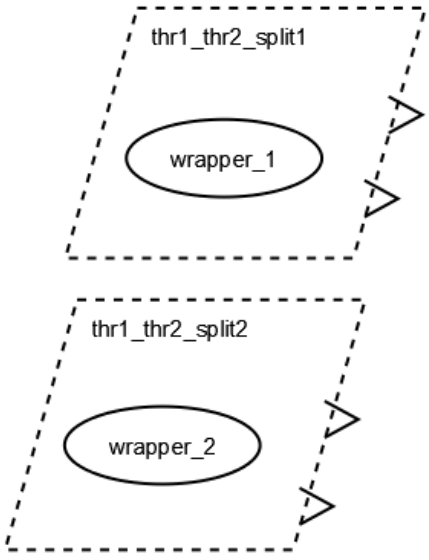


FIG. 3.24 – Split d'un thread contenant 2 call sequences

Pour pallier ce problème, il faudrait pouvoir spécifier le comportement de chaque thread. Cela pourrait être fait grâce à l'annexe comportementale du langage AADL. Celle-ci, cependant,

n'était pas disponible au moment de la conception des opérations d'optimisation.

3.6 Conclusion

L'exemple de l'opération *Split* nous indique qu'il peut être nécessaire, à l'avenir, d'ajouter des nouvelles opérations d'optimisation. Nous avons développé dans notre travail de thèse une approche générique de l'optimisation, qui permet d'intégrer simplement ces dernières. L'ajout d'une opération doit suivre un certain nombre d'étapes :

- la spécification de la traduction dans le modèle AADL, sur le modèle du travail effectué dans ce chapitre ;
- la spécification de l'impact de cette nouvelles opération, également sur le modèle du travail effectué dans ce chapitre ;
- la traduction de cet impact dans différents niveaux d'évaluation ainsi que l'intégration dans l'algorithme d'optimisation, comme illustré sur les opérations Merge et Move dans le chapitre 4 ;
- l'implantation des fonctions d'évaluation spécifiques à ces opérations, en utilisant le formalisme décrit dans le chapitre 5 ;
- l'implantation des opérations d'évaluation dans l'outil ocarina, en suivant le modèle décrit dans le chapitre 6.

Dans ce chapitre, nous avons présenté les opérations permettant d'améliorer certains aspects des performances du système réel. Dans le chapitre suivant, nous présentons comment piloter l'application de ces opérations d'une façon proche de l'optimalité. Nous montrerons pour ce faire plusieurs algorithmes, puis nous proposerons une étude des différents critères de performance, de la façon dont ils peuvent être mesurés et comment nous pouvons intégrer ces solution d'évaluation dans le contexte d'une optimisation dirigée par les modèles, ainsi que les bénéfices en terme d'automatisation que nous pouvons en attendre.

4

Évaluation des modèles

Nous montrons dans ce chapitre comment utiliser les opérations décrites dans le chapitre 3 pour optimiser le système réel. Dans la première section, nous présentons une classification du problème présenté par l'optimisation sur les modèles, discutons des solutions proposées dans la littérature, puis proposons notre propre solution. Dans la seconde section, nous présentons précisément les algorithmes d'optimisation utilisés. La troisième section traite des possibilités d'évaluation des systèmes temps-réel embarqués et des critères pertinents pour ce faire. Finalement, la dernière section présente l'apport concret offert par l'ingénierie des modèles à ce processus, ainsi que l'emplacement exact de celui-ci dans la chaîne d'optimisation.

4.1 Optimisation et évaluation

Les techniques d'optimisation présentées dans le chapitre 3 permettent de donner un nouveau découpage aux sous-programmes que le découpage initial (faisant correspondre un sous-programme à un thread), ainsi que de procéder à un déplacement de charge. Pour diriger ces opérations, plusieurs solutions algorithmiques sont possibles. Dans cette section, nous commençons par donner une classification des différents éléments qui composent notre problème, puis nous évaluons l'intérêt de trois d'entre elles : l'utilisation d'heuristiques simples, d'algorithme évolutifs ou de techniques d'optimisation de graphe. Nous décrivons ensuite comment chacun de ces types de méthode peuvent être utilisés et finalement quels sont leurs avantages et défauts respectifs, comme nous l'avons présenté dans [54].

4.1.1 Classification du problème

Les opérations proposées dans le chapitre 3 consistent à trouver l'ensemble de combinaisons de tâches (call sequences) parmi les threads optimales selon les critères considérés. On peut associer à chaque opération un coût et une valeur (déduits de la variation de performance du système final). On retrouve dans cette situation une variante du problème bien connu du sac à dos, qui consiste à placer un nombre maximal (ou une valeur maximale) de composants auxquels on associe des poids disctincts dans un conteneur de capacité finie. On pourra noter que cette caractérisation n'est pas propre à nos méthodes d'optimisation : indépendamment de la granularité choisie pour l'optimisation, celle-ci s'appuie nécessairement sur un réarrangement des éléments du système.

Nous avons vu également que les associations ne sont pas équivalentes pour un modèle et une opération donnés. Par exemple, en supposant qu'un thread soit constitué d'un ensemble de call sequences non-vides, les performances obtenues par une nouvelle opération $Merge$ varieront

selon le thread avec lequel le thread initial sera fusionné. Il en découle qu'on ne peut pas attribuer un poids fixe à un élément candidat au placement comme c'est le cas dans le problème du sac à dos général, mais que le calcul du poids (ou de la valeur) d'une opération doit prendre en compte l'état courant du modèle. Ce type de problème correspond au cas particulier des sacs à dos multi-quadratiques.

Finalement, le problème du sac à dos quadratique peut être associé à celui de la recherche d'un nombre minimal de cliques dans le graphe correspondant au système. Des algorithmes exacts — mais non polynomiaux — existent pour trouver des solutions à des exemples de taille restreinte de ces problèmes [33, ?]. Une telle approche est donc possible également dans notre cas.

4.1.2 Implémentations des techniques d'optimisation

Heuristique De nombreux travaux ont été effectués dans la résolution de ce problème NP-complet [65]. Les auteurs de [64] présentent un algorithme permettant d'obtenir une solution proche de la solution optimale en temps polynomial. Une propriété intéressante de cet algorithme est de séparer nettement la partie évaluation de la partie optimisation. Ce découpage nous permet de rendre paramétrable les critères d'évaluation. Pour être efficace, cette approche nécessite cependant de pouvoir évaluer a priori les effets d'une opération d'optimisation donnée sur le modèle.

Algorithmes génétiques Les mêmes auteurs proposent également une technique basée sur les algorithmes génétiques pour trouver une valeur proche de l'optimale. Une telle approche permet, comme dans le cas précédent, de séparer strictement la partie évaluation de la partie optimisation. La première sera représentée par la fonction de *fitness* de l'algorithme génétique, qui évaluera la valeur du système courant. Une solution peut être représentée par un tableau de la taille du nombre de tâches à accomplir, dans chaque case duquel on indique le thread correspondant. Les opérations *Merge* et *Move* sont utilisées pour implémenter les fonctions de mutation et de croisement. Un aspect intéressant de cette approche est de ne pas avoir besoin d'hypothèses sur les effets d'une opération d'optimisation donnée, ainsi que d'éviter les maximum locaux sans toutefois explorer la totalité des solutions.

Séparation-évaluation Finalement, les auteurs de [34] proposent une solution efficace basée sur une stratégie séparation-évaluation — c'est-à-dire en divisant le problème en une série de problèmes similaires plus petits — pour calculer la solution optimal du problème du sac à dos quadratique ou de la clique maximale. Pour cela, il faut disposer d'un moyen de calcul d'une borne inférieure (ou supérieure) pour un ensemble de solutions ainsi que d'une fonction permettant de diviser l'espace de recherche. La borne est généralement en fonction du coût des solutions présents dans l'espace.

4.1.3 Discussion

Les auteurs de [64] fournissent des tests sur lesquels les algorithmes génétiques sont significativement plus longs à s'exécuter que la solution exacte, pour un gain d'efficacité marginale (la solution finale obtenue est supérieure de 0,5% à celle obtenue par une simple heuristique gloutonne).

Dans le cas de notre problème d'optimisation, il est important de noter que les solutions sont des ensembles d'opérations d'optimisation, et que le coût des opérations des algorithmes

génétiques (croisement, mutation) est important (il s'agit à chaque fois d'opérations profondes sur les modèles, comme montré dans le chapitre 3).

Trouver une solution optimale par une opération de séparation-évaluation présente plusieurs difficultés. En premier lieu, le choix d'une borne minimale est possible dans un contexte de sac à dos quadratique, mais s'applique mal au sac à dos multi-quadratique — puisque une solution très éloignée de l'optimal peut rendre possible la constitution d'autres cliques dans le système. Dans le cas qui nous interesse, il est très difficile de calculer une fonction de coût (ou de valeur), autre que heuristique, sur des opérations non effectuées. De plus, les solutions performantes au niveau algorithmique utilisent souvent des matrices de solutions qui représentent dans notre problème un espace mémoire très important.

Dans ce chapitre, nous commençons par présenter nos algorithmes d'optimisation, basés sur des techniques purement heuristiques, puis nous discutons des différents critères d'évaluation. Finalement, nous montrons comment nous pouvons assembler ces deux étapes au sein d'un processus d'optimisation complet.

4.2 Algorithmes pour diriger l'optimisation

Nous avons développé deux algorithmes pour diriger l'optimisation des systèmes TRE. Tous deux s'appuient sur les travaux présentés dans [64], mais diffèrent par leur complexité. Bien que tous deux reposent sur une stratégie gloutonne, l'algorithme `Full Greedy Heuristic` (FGH) utilise notre connaissance des effets de chaque opération pour limiter le nombre d'états du système à explorer, tandis que l'algorithme `Half Greedy Heuristic` (HGH) se contente d'utiliser une stratégie gloutonne plus classique, explorant un plus grand nombre d'états est donc moins susceptible de tomber dans un minimum local.

Dans les deux cas, les fonctions d'évaluation et de coût sont paramétriques. Une discussion sur les critères d'évaluation peut être trouvée plus loin en section 4.3. Cet ensemble proche de l'optimal est ensuite fusionné, et une opération move est sélectionnée au sein de l'ensemble des *Move* possibles depuis le process où la fusion a été effectuée vers un process tiers. La sélection est faite selon une fonction d'évaluation elle aussi paramétrique.

4.2.1 Full-Greedy Heuristic

La solution FGH présentée par le pseudo-code 1 consiste à élire un thread selon les critères définis dans le chapitre 3, puis à chercher l'ensemble de threads sur lequel l'opération `Merge` produit le système de plus haute valeur selon la fonction d'évaluation considérée — cet ensemble est représenté par la variable *candidate_set*. Pour cela, elle utilise l'opération *Compute_Value*, qui calcule le gain espéré de l'ajout d'un thread (second paramètre) à l'ensemble de threads passé en premier paramètre. Comme la liste des threads est triée selon le critère qui a servi à élire le thread d'origine, cette solution tend à converger rapidement vers un ensemble optimal.

Finalement, l'algorithme se répète tant qu'il reste un ensemble de threads à fusionner non vide à valeur positive.

Dans cet algorithme, nous faisons trois boucles sur la liste des threads, une itération de la boucle la plus extérieurs étant nécessairement terminée par une opération *Merge* sur au moins deux threads distincts. La taille de la liste décroît donc au minimum de un élément par itération. La complexité théorique en terme d'opérations d'évaluation de cet algorithme est donc en $O(n^3)$, où n est le nombre de threads dans l'ensemble du système. Il est à noter que, selon notre fonction d'évaluation, les déplacements de threads tendent à égaliser la taille des process quant au nombre de threads à fusionner, et donc fait baisser la complexité effective de l'algorithme. Ce résultat

```
Input: System S
forall p ∈ Process(S) do
    repeat
        Sort(Threads(p))
        T ← First(Threads(p))
        Candidate_Set ← ∅
        repeat
            Best_Value ← 0  forall t2 ∈ Threads(p) do
                if t2 ≠ T then
                    Current_Value ← Compute_Value(Candidate_Set, t2) if
                    Current_Value > Best_Value then
                        Best_Value ← Current_Value Best_Candidate ← t2
                    end
                end
            end
            Candidate_Set ← Candidate_Set ∪ Best_Candidate
        until noBest_Candidatefound ;
        if Candidate_Set ≠ ∅ then
            S ← Merge(Candidate_Set) S ← Move(Candidate_Set, S)
        end
    until Candidate_ Set = ∅ ;
end
```

Algorithm 1: Full-Greedy Heuristic

n'est cependant pas généralisable, et illustre bien l'importance de la fonction de sélection. Deux autre éléments tendent encore à faire baisser la complexité effective de l'algorithme :

- les ensembles de threads à fusionner tendent à contenir plus de deux threads ;
- la fonction de coût empêche l'exploration de nombreuses solutions impossible à atteindre en pratique.

4.2.2 Half-Greedy Heuristic

La solution HGH présentée par le pseudo-code 2 consiste à chercher l'ensemble de `Merge` contigus produisant le système de plus haute valeur selon la fonction d'évaluation considérée. En cas de réussite, un *Move* est effectué dans les mêmes conditions que dans l'algorithme FGH. L'algorithme est itéré jusqu'à ce qu'aucun ensemble de valeur positive ne soit trouvé.

Comparativement à l'algorithme FGH, HGH cherche donc tous les ensembles possibles sans préjuger d'un élément de départ. Dans cet algorithme, quatres boucles sont effectuées dans le pire des cas jusqu'à ce que la liste des threads soit réduite à un unique élément. Comme dans le cas de l'algorithme 2, un *Merge* précède nécessairement une nouvelle itération de la boucle la plus extérieure. Sa complexité est donc supérieure à celle de la précédente, de l'ordre de $O(n^4)$, où n est le nombre de threads dans l'ensemble du système. Pour les mêmes raisons que FGH, cependant, sa complexité effective est généralement bien inférieure.

```
Input: System S
forall p ∈ Process(S) do
    repeat
        Best_Set_Value ← 0
        Final_Set ← ∅
        forall T ∈ Threads(p) do
            T ← First(Threads(p))
            Candidate_Set ← ∅
            repeat
                Best_Value ← 0
                forall t2 ∈ Threads(p) do
                    if t2 ≠ T then
                        Current_Value ← Compute_Value(Candidate_Set, t2) if
                        Current_Value > Best_Value then
                            Best_Value ← Current_Value Best_Candidate ← t2
                        end
                    end
                end
                Candidate_Set ← Candidate_Set ∪ Best_Candidate
            until noBest_Candidatefound ;
            if Best_Value > Best_Set_Value then
                Best_Set_Value ← Best_Value Final_Set ← Candidate_Set
            end
        end
        if Final_Set ≠ ∅ then
            S ← Merge(Final_Set) S ← Move(Final_Set, S)
        end
    until Final_Set = ∅ ;
end
```

Algorithm 2: Half Greedy Algorithm

4.3 Contraintes sur les systèmes TRE

Nous avons vu que la détermination d'une fonction d'évaluation à la fois précise et efficace est cruciale pour permettre la justesse et la performance des algorithmes d'optimisation. Nous présentons ici les contraintes qui s'appliquent aux systèmes TRE, ainsi que des heuristiques permettant de mesurer le degré ou la probabilité de violation de ces contraintes, tant au niveau du modèle qu'à celui des opérations.

4.3.1 Ordonnancement

Par définition, un système temps-réel doit être strictement ordonnançable. La technique d'ordonnancement utilisée (RMA, EDF, FIFO...) fait varier la taille du sous-ensemble des systèmes ordonnançables, mais visent tous à remplir la condition de l'ordonnancement : assurer que, quelque soit l'exécution, toutes les tâches puissent être exécutées avant leur échéance.

Le problème de l'ordonnancement met en jeu deux aspects différents : d'une part, l'analyse du pire cas d'exécution (WCET), puis le calcul du temps de réponse des différentes tâches. Le second aspect utilise les informations produites par le premier, mais ne s'y limite pas. Nous verrons dans cette partie quels sont les problèmes posés par ces deux aspects, et comment nous pouvons les résoudre dans le cadre d'une approche outillée et dirigée par les modèles.

Analyse du WCET

Le WCET peut être calculé de façon statique, ou sa valeur peut être mesurée. Une troisième approche consiste à utiliser de façon complémentaire les deux approches précédentes. Les méthodes impliquant des mesures, si elles donnent des valeurs très proches du WCET réel, présentent un risque de sous-estimation qui les disqualifie pour les systèmes temps-réels durs [90]. Nous nous limiterons donc à présenter les problématiques liées à l'évaluation statique.

De nombreux travaux ont été effectués pour couvrir l'analyse statique du WCET. Celle-ci, en effet, présente une difficulté intrinsèque : les architectures matérielles modernes sont de moins en moins prédictibles, essentiellement pour deux raisons : les caches mémoire, et le pipelining dans les processeurs. A ces deux problèmes s'ajoute celui de l'analyse du flot de contrôle, qui permet de déterminer quels chemins d'exécutions sont possibles.

Cache Le cache permet de réduire les temps de lecture et — dans une moindre mesure — d'écriture pour les données ou les instructions en mémoire. Ce gain, cependant, n'est pas homogène : le cache doit charger les données en mémoire en cas de miss. Un enjeu important est donc de déterminer, dans le code fourni, quels éléments amèneront à un miss sur une architecture donnée. L'analyse de cet aspect du WCET est nommé *value analysis*, et peut offrir une image de la plupart des valeurs dans le cache et en mémoire sous l'hypothèse d'un code statique et respectant certaines contraintes [87].

Pipelining Le problème de l'influence du pipelining est regroupé avec celui du cache sous l'appelation *analyse du comportement processeur*. Le premier a pourtant des implications particulières, puisqu'il permet d'appliquer statistiquement un facteur multiplicateur à l'exécution des instructions. Cette accélération est cependant remise en cause en raison des dépendances que peuvent avoir les différentes instructions entre elles, et par les branchements conditionnels. Il importe donc de tenir compte de ces éléments pour calculer la vitesse d'exécution effective d'un ensemble d'instructions. Pour cela, il faut produire un modèle temporel du fonctionnement du

processeur, qui ne peut être obtenu que par une étude très détaillée de celui-ci, et qui permet de suivre l'état du processeur tout au long de l'exécution [57].

Flot d'exécution Le calcul du flot d'exécution revient à construire le graphe des branchements et des appels. Quand il est effectué sur du code-source, il est difficile d'en déduire les chemins d'exécution effectifs, car l'étape de la compilation (voire le processeur lui-même) va généralement procéder à des réordonnancements sur le code — c'est pourquoi le flot d'exécution est généralement calculé sur le code machine. Il est cependant très difficile d'inférer le chemin d'exécution à partir du code machine. Les auteurs de [55] proposent une méthode sur le code intermédiaire. Cette méthode utilise une analyse de la valeur des variables associées au compteur de la boucle, pour des compilateurs communs dans l'embarqué. En pratique, les outils existants exigent généralement des indications fournies par l'utilisateur pour calculer les flots d'exécutions. Ce besoin de configuration empêche l'automatisation complète de l'évaluation du WCET.

Calcul du temps de réponse

Le calcul du temps de réponse des tâches est un corollaire de l'ordonnancement, puisqu'une des méthodes les plus courantes pour vérifier un ordonnancement est d'effectuer une analyse du temps de réponse de chaque tâche dans le pire cas (*Response Time Analysis*), puis de vérifier que les échéances des différentes tâches sont respectées. Le calcul du temps de réponse dépend de la méthode d'ordonnancement sélectionnée [42]. Une telle opération demande d'avoir une vision de l'ensemble de l'application — puisque les tâches peuvent être en attente du complétement d'autres tâches, et donc concernées par leur temps de réponse. Ce calcul va prendre en entrée les WCETs des différentes tâches. Les auteurs de [36] proposent une méthode pour calculer en temps linéaire des bornes supérieures pour les différentes tâches du système, dans un système à priorités fixes. On remarquera qu'un tel calcul demande la connaissance d'information architecturales, telles que les dépendances entre les différentes tâches ou leur priorité.

4.3.2 Dimensionnement

Les systèmes TRE se doivent de respecter les contraintes de dimensionnement, pour assurer leur embarquabilité. L'empreinte mémoire des différents processes, en particulier, doit respecter les limites imposées par l'architecture matérielle. D'autres contraintes de dimensionnement peuvent également s'appliquer, par exemple sur les mémoires spécifiques, le nombre de verrous matériels utilisés ou le volume des connexions inter-processus. Dans cette étude, nous nous limiterons cependant à l'empreinte mémoire.

Taille de la pile

La pile contient les informations liées à la gestion des appels, en particulier les valeurs des différents paramètres, ainsi qu'un certain nombre de variables de gestion dont le rôle exact dépend de l'architecture matérielle. Puisqu'il dépend directement des différents appels de sous-programmes, le problème du calcul de la pile est donc similaire à une version simplifiée du problème du calcul du WCET qui se réduirait à déterminer les flots de contrôle [88].

Taille des données

L'analyse de la taille des données comporte deux aspects : d'une part l'analyse des variables déclarées statiquement (segments de mémoire `data` et `BSS`) et d'autre part l'analyse des données

allouées dynamiquement (le tas). La première partie est simple à effectuer, que ce soit au niveau du code source (on peut simplement sommer les déclarations de variables) ou du code machine (où ces segments apparaissent de façon disctincte du reste du code). Le problème de l'analyse de la taille du tas est pour sa part connu pour être difficile. Celui-ci dépend de l'architecture matérielle, du compilateur et, comme pour l'analyse du WCET, doit prendre en compte les flots d'exécution. Une technique pour calculer la borne supérieure de ce dernier a été proposée par les auteurs de [30].

4.3.3 Sûreté

En plus des aspects liés au temps-réel et à l'embarqué, des contraintes peuvent également découler du contexte éventuellement critique dans lequel le système s'exécute.

La sûreté dans les systèmes informatiques a été traitée de nombreuses façons. Une des approches propose la stricte séparation des partitions, concept venu de celui de *Robust Partitioning* employé pour la sûreté dans la norme *Integrated Modular Avionics* (IMA [66]). Cette approche, cependant, soulève la question de l'échange d'informations sécurisé. C'est à cet effet qu'a été conçue l'approche *Multiple Independant Levels of Security* (MILS [82]). Il s'agit, en utilisant des niveaux de criticité sur les connexions et les partitions, d'offrir l'assurance qu'une défaillance dans une partition n'aura pas d'impact sur les partitions de criticité supérieure. Une telle approche permet de contenir strictement les défaillances, et donc de faire tourner plusieurs processes sur le même processeur. Cette technique était utilisée plus anciennement pour assurer la sécurité du système, comme par exemple dans [50].

4.4 Evaluer une architecture

L'évaluation des critères définis dans la section précédente peut être effectuée à plusieurs niveaux. En premier lieu, il est possible de chercher à évaluer le *système réel*, c'est-à-dire à la fois une implémentation, une configuration et un déploiement. Une telle approche, si elle a l'intérêt d'assurer une précision maximale, est cependant longue et difficile à effectuer, car la chaine de transformations qui ont eu lieu entre les spécifications initiales et le code machine (implémentation, compilation...) a généralement causé des pertes d'informations (flots de contrôle ou de données) et en a occulté d'autres (contraintes non-fonctionnelles). Enfin, il est fréquent que des besoins d'analyse apparaissent avant que le système réel soit implémenté, configuré ou déployé.

Une solution consiste à utiliser ensemble les spécifications initiales et les informations issues de l'analyse du système réel. Si une telle solution résout le problème de la perte d'information, elle laisse entière celle de la durée de l'analyse et de la non-existence du système réel.

Pour palier ce problème, certaines analyses préliminaires peuvent se contenter d'analyses a priori, en calculant la valeur des critères en fonction de leur valeur dans le système généré à partir du modèle antérieur. Dans le cas d'un processus d'optimisation où l'impact des différentes opérations peut être évalué a priori (comme montré dans le chapitre 3, et où l'évaluation est déclenchée fréquemment, une telle approche est particulièrement adaptée, car elle garantit à la fois une plus grande vitesse de calcul et une relative indépendance par rapport à la disponibilité du système réel correspondant au modèle courant.

Une troisième approche consiste à exploiter plus avant notre connaissance de l'impact des opérations effectuées (ici, opérations d'optimisation) pour calculer non plus la valeur d'un certain critère, mais la variation de cette valeur que va induire l'opération considérée. Cette méthode est la plus rapide, puisqu'elle n'explore qu'un sous-ensemble du modèle jugé sensible à l'opération, au prix d'une certaine perte de précision.

4.4.1 Le calcul des performances avec AADL

Ordonnancement

Analyse du WCET Nous avons vu qu'il existait des approches pour calculer le WCET en faisant les trois hypothèses suivantes :

1. en amont, une discipline de code stricte (pas de récursion, pas d'allocation dynamique, etc.) pour le développeur ;
2. la connaissance des motifs de codes générés à partir des modèles AADL ;
3. en aval, une intervention manuelle pour la configuration de l'outil.

En d'autres termes, nous pouvons effectuer une analyse semi-automatique du WCET de code source discipliné, au moyen d'outils tels que aiT [3] [86] ou Bound-T [4] [58].

La plate-forme de génération Ocarina, sur laquelle nous travaillons et à laquelle nous avons collaboré — et dont les principaux concepteurs appartiennent à notre laboratoire — peut assurer la stricte discipline du code (c'est-à-dire le déterminisme de son exécution) en employant deux moyens :

- un profile pour le code source *Ada*, Ravenscar [44], dont le respect est assuré par le compilateur GNAT ;
- une plateforme d'exécution logicielle, PolyORB-HI [92], suivant les mêmes préceptes.

Cette approche est résumée par les auteurs de [59], qui précisent également les aspects optimaux par construction du code généré *en fonction d'un modèle donné*, nous autorisant ainsi à poursuivre le travail sur l'optimisation en concentrant nos efforts sur l'optimisation des modèles.

La configuration de l'outil d'analyse implique la connaissance de plusieurs informations non-fonctionnelles, telle que la périodicité d'une tâche ou le type du processeur. Ces éléments sont présents dans le modèle architectural. Disposer d'un langage d'expression de contraintes et de requêtes sur le modèle architectural nous permettrait donc de spécifier ces informations.

Certaines fonctions bas-niveau générées par PolyORB-HI résistent à l'analyse du WCET, généralement à cause de boucles dont l'arrêt est contrôlé par une expression complexe, mettant en jeu des éléments du modèle. Un exemple d'une telle fonction de PolyORB-HI est celle permettant de récupérer les messages reçus. Nous savons qu'elle va se répéter pour chaque thread connecté au thread courant. Au niveau code, il n'est pas possible d'inférer la valeur de cette information. Au niveau du modèle, il s'agit d'une opération relativement triviale si l'on possède un langage d'expression de requête sur le modèle AADL.

Dans le cadre de notre thèse, nous avons en effet choisi l'outil Bound-T. Les raisons qui ont guidé ce choix sont le support des architectures de type SPARC (ERC32, utilisé dans le cadre de l'industrie spatiale). D'autres choix auraient bien sûr été possibles, et pourraient être utilisés dans des implantations ultérieures. Nous présentons dans le chapitre 5 comment nous interrogeons un modèle AADL, puis nous montrons dans la section 6.2 comment nous utilisons notre connaissance des motifs de génération de code pour configurer un outil d'analyse de WCET, comme nous l'avons exposé en [52].

Calcul du temps de réponse Nous avons vu que le calcul du temps de réponse d'une tâche demandait la connaissance d'information non-fonctionnelles et architecturales. Ces informations sont présentes dans le modèle AADL : les auteurs du langage ont d'ailleurs expliqué précisément dans [51] une méthode pour calculer le temps de réponse de chaque flot d'exécution (en indiquant

[3] http://www.absint.de/ait/
[4] http://www.tidorum.fi/bound-t/

les propriétés à utiliser, les opérations à effectuer, etc.). Cette approche est implantée dans la plateforme OSATE [5]. Les auteurs de l'outil Cheddar [6] proposent en [83] une approche pour arriver sensiblement au même résultat (RTA, ordonnançabilité RMA, EDF, POSIX 1003b, etc.).

Il est donc possible, si l'on connait les informations non-fonctionnelles déduites des spécifications (périodicité et échéance, disponibles dans le modèle), de l'analyse du WCET (voir ci-dessus), et de la topologie (également disponible dans le modèle), de déduire l'ordonnançabilité du système et le temps de réponse des différents flots d'exécution. Dans notre étude, nous nous sommes limités à vérifier l'ordonnançabilité en suivant l'ordonnancement RMA grâce à l'outil Cheddar.

Dimensionnement

Nous avons vu que l'empreinte mémoire dépendait de trois éléments : taille des données statiques, taille du tas et taille de la pile. Le premier élément peut aisément être déduit de l'analyse du code-source ou des binaires. Dans le cadre de notre travail de thèse, nous avons utilisé l'outil GNU size.

La taille du tas dépend uniquement des allocations dynamiques effectuées lors de l'exécution. Dans les systèmes critiques, il est commun d'interdire toute allocation dynamique de mémoire, celle-ci étant suceptible d'introduire des défaillances et des temps de réponse très difficiles à prédire. Le profil Haute Intégrité pour Ada suit un telle politique [62] : la taille du tas n'a donc pas à être calculée dans le cadre de notre travail.

Finalement, le calcul de la taille de la pile est le seul élément complexe à calculer. Nous avons montré la similarité de ce problème avec celui de l'analyse du WCET et, plus généralement, de l'exploration des flots d'exécution. Cette similarité explique que les outils d'analyse du WCET étudiés permettaient également d'effectuer une analyse de la pile. On notera que, dans ce cas, les conditions nécessaires pour l'analyse sont plus simples : aucune hypothèse n'est faite sur la nature du code. Il est cependant nécessaire de renseigner la taille des piles de certaines fonctions de bas niveau — comme dans le cas de l'analyse du WCET, ces informations peuvent être retrouvées dans le modèle AADL. Originellement, la connaissance de la plateforme d'exécution logicielle (intergiciel et système d'exploitation) permet de connaitre ces valeurs.

Sûreté

Nous avons donné un exemple de vérifications possibles sur la sûreté d'une application. Une première observation est que cette mesure ne peut être, par nature, que booléenne. Il s'agit donc non pas de mesurer, mais de valider un système. Cette validation doit être effectuée (au moins) sur deux aspects distincts : le modèle et le code source.

Pour valider le modèle, nous pouvons utiliser les informations extraites du modèle architectural pour vérifier des contraintes relevant des aspects topologiques et non-fonctionnels de l'application, si nous avons un moyen d'exprimer celles-ci. Nous donnons dans le chapitre 5 un exemple des propriétés à vérifier sur un système AADL partitionné, dont le principe est expliqué par les auteurs en [39].

Nous devons ensuite vérifier que le code source généré ne présente pas de risque de défaillance non-maîtrisable. Pour cela, comme expliqué ci-dessus, nous adoptons le code Ada et son profil Ravenscar, ainsi que l'intergiciel PolyORB-HI. Ce dernier a été formellement prouvé contre un certain nombre de défaillances (absence d'interbloquages entre autres) [60], et respecte le profil

[5] http://www.aadl.info/aadl/currentsite/tool/osate.html
[6] http://beru.univ-brest.fr/~singhoff/cheddar/index-fr.html

Ravenscar ainsi que le profil Haute Intégrité pour Ada, assurant ainsi une sûreté de bout en bout du code généré.

4.4.2 Niveau d'évaluation

Au regard des considérations précédentes, nous avons défini trois niveaux différents d'évaluation pour les systèmes TRE :

- **Evaluation Basée sur les Binaires** (EBB), qui évalue le critère à la fois sur le système réel et sur son modèle ;
- **Evaluation Basée sur les Modèles** (EBM), qui évalue le critère sur le modèle ;
- **Evaluation Basée sur les Opérations** (EBO), qui évalue la variation du critère pour une opération sur le modèle.

Comme nous l'avons vu, les méthodes d'évaluation rapides (EBO) utilisent des heuristiques et font donc perdre en précision à l'évaluation. Pour éviter que, au fil des optimisations, l'évaluation du modèle diverge excessivement du système réel, nous avons besoin d'appeler périodiquement les méthodes de EBM voire de EBB. Par ailleurs, certains critères comme l'ordonnançabilité ne peuvent être évalués localement. Cependant, pour que notre évaluation soit efficace en terme de temps d'exécution, nous devons offrir une alternative plus rapide à ces niveaux d'évaluation, et devons donc majoritairement utiliser la EBO. Les trois niveaux sont donc nécessaires conjointement à un processus d'optimisation réaliste.

4.4.3 Ordonnançabilité

Évaluation Basée sur les Modèles

Une limitation intrinsèque de l'ordonnancement est la distance entre le pire temps de réponse (WCRT) et l'échéance. Pour un ensemble de threads, plusieurs indicateurs peuvent donc être utilisés. Nous les présentons ci-après, en utilisant les notations suivantes :

- $T_{\sqcup}$ est la date de déclenchement de la tâche associée au thread $\sqcup$;
- $C_{\sqcup}$ est le pire temps d'exécution de la tâche associée au thread $\sqcup$;
- $D_{\sqcup}$ est l'échéance de la tâche associée au thread $\sqcup$.

Nous proposons donc les heuristiques suivantes :

- **distance minimum à l'échéance**

$$Min_{\delta_{wcet}} \leftarrow Min(D_{\mathcal{T}} - (T_{\mathcal{T}} + C_{\mathcal{T}}))$$

- **distance moyenne à l'échéance**

$$Avg_{\delta_{wcet}} \leftarrow \sum(D_{\mathcal{T}} - (T_{\mathcal{T}} + C_{\mathcal{T}}))/cardinal(Threads)$$

- **somme des distances à l'échéance**

$$Sum_{\delta_{wcet}} \leftarrow \sum(D_{\mathcal{T}} - (T_{\mathcal{T}} + C_{\mathcal{T}}))$$

Évaluation Basée sur les Opérations

Dans le cas de l'évaluation de l'impact d'une fusion donnée entre un certain nombre de threads, il existe également plusieurs possibilités. L'une d'elles consiste à calculer la variation entre la distance globale à l'échéance ($Min_{\delta_{wcet}}$, $Avg_{\delta_{wcet}}$ ou $Sum_{\delta_{wcet}}$) avant et après le merge.

Une autre solution, moins précise, se base sur la connaissance du fonctionnement des Merge. Nous savons que le thread résultant va avoir une période égale au plus grand dénominateur

commun des threads fusionnés. Il est donc possible d'avoir une indication sur la nouvelle échéance du thread produit en fonction des échéances des threads candidats au Merge, contenus dans l'ensemble T : cette méthode a l'avantage de ne pas avoir à évaluer tout le modèle, et est donc plus rapide, mais ne tient pas compte du WCET, et perd donc en précision.

Finalement, nous pouvons utiliser la connaissance que nous avons de l'impact des Merge pour évaluer le nouveau WCET des threads. Nous avons vu que le principal effet de la fusion est de réduire le nombre de connexions inter-threads, et que cette opération permet de réduire le WCET. Une heuristique simple permettant d'obtenir la fusion la plus avantageuse pour l'ordonnancement est donc de maximiser le nombre de connexion supprimées.

4.4.4 Dimensionnement

Les systèmes TRE se doivent de respecter les contraintes de dimensionnement, pour assurer leur embarquabilité. L'empreinte mémoire des différents processes, en particulier, doit respecter les limites imposées par l'architecture matérielle. D'autres contraintes de dimensionnement peuvent également s'appliquer, par exemple sur les mémoires spécifiques, le nombre de verrous matériels utilisés ou le volume des connexions inter-processus. Dans cette étude, nous nous limiterons cependant à l'empreinte mémoire.

Évaluation Basée sur les Modèles

La différence entre l'empreinte mémoire et la mémoire disponible donne une bonne indication d'efficacité. Comme pour l'évaluation de l'ordonnançabilité, nous pouvons décliner cet indicateur sur plusieurs processeurs en retournant le minimum, la moyenne ou la somme de ces différences.

Évaluation Basée sur les Opérations

La réduction de mémoire est linéaire avec le nombre de Merge effectués. Il est donc impossible de fournir une heuristique qui distingue deux *Merge* sans se référer au modèle. La seule heuristique pertinente est ici la différence entre l'empreinte mémoire du process contenant le thread et la mémoire physique associée. Dans le cas du *Move*, la valeur de l'opération est celle de l'empreinte mémoire du thread déplacé. Nous avons vu plus haut comment mesurer cette valeur.

4.4.5 Sûreté

Pour répondre à ce critère, nous ne pouvons pas définir a priori de règle générale : les contraintes de sûreté dépendent largement de la stratégie employée par l'architecte et du contexte. En règle générale, ces contraintes peuvent être vérifiées lors de la EBM.

Dans notre mémoire, nous proposons deux types de vérifications liées à la sûreté du système :

- **vérification des critères de sûreté définis dans le standard Ravenscar** ;
- **vérification de l'intégrité des connexions dans le cas d'un système distribué**.

La définition et la mise en œuvre de ces critères sont illustrées dans le chapitre 5. Le fait que les contraintes liées à la sûreté soient spécifiques à une approche souligne l'importance de laisser à l'architecte la possibilité de définir ses propres contraintes.

4.4.6 Pondération des critères d'évaluation

Il est possible de donner aux différents critères d'évaluation une importance variable selon qu'ils soient ou non satisfaits par le modèle courant. En d'autres termes, il est possible de déduire

la pondération des différents critères de l'architecture logicielle et matérielle telle qu'elle est spécifiée dans le modèle.

Une illustration d'une telle possibilité se retrouve dans le cas de critères non-pertinents dans certains domaines, par exemple la consommation énergétique dans l'automobile — où la rotation des roues du véhicule produit l'énergie qui permet de faire fonctionner ses différents systèmes embarqués. L'absence d'une contrainte peut être déduite de l'absence de la propriété servant à la mesurer. Dans ce cas, le poids du critère associé dans la fonction linéaire calculant la valeur du modèle doit être mis à zéro.

De façon plus générale, plusieurs politiques sont possibles quant au calcul du poids de chaque critère. Toutes demandent de calculer initialement une ou plusieurs valeurs-seuils qui correspondent à la "satisfaction" du critère pour le modèle courant. Le poids d'un critère doit alors augmenter à mesure que la valeur du critère pour un modèle baisse en-dessous des différentes valeurs-seuil, et baisser quand elle les dépasse. On peut éviter les oscillations autour d'une valeur en gardant une trace des états précédents.

De façon générale, cette approche est relativement difficile, car demandant un travail important de la part du développeur. Il s'agit d'ailleurs d'une problématique souvent abordée dans le domaine de l'optimisation, par exemple par [23], qui propose également une approche par seuil — bien que sans notion de pondération — pour l'évaluation multi-critère. Si la détermination du seuil peut être automatisée grâce à un langage de description de contraintes (cf. chapitre 5), la valeur des poids associés à chaque intervalle doit être affectée manuellement. Une solution pour résoudre ce problème pourrait être l'application initiale d'une méta-heuristique de type algorithme évolutif, dont les résultats (en termes de pondération) pourraient ensuite être réutilisés dans d'autres modèles.

4.4.7 Pipeline d'évaluation

Pour pouvoir mettre en œuvre un processus d'évaluation, il est nécessaire de garantir que les informations de l'évaluation soient conservées et puissent être interprétées lors d'évaluations ultérieures. Dans le cas de la EBB en particulier, des informations sont obtenues par des procédés tiers (outils d'analyse du WCET, de la pile, de l'ordonnancement, etc.), générant des informations utiles pour des analyses ultérieures (borne supérieure sur le WCET et la pile de sous-programmes, priorités des différents threads, etc.).

Il est donc nécessaire que notre langage de modélisation et les outils qui permettent de le manipuler autorise l'ajout d'informations diverses sur le modèle d'origine. AADL et ses `property` offre cette possibilité. Dans le reste de ce mémoire, nous parlerons de *modèle enrichi* pour décrire le modèle auquel ces informations ont été ajoutées.

Le schéma 4.1 illustre un processus d'évaluation organisé autour du modèle architectural. Les composants en gris foncé sont les modules d'évaluation correspondant aux différents niveaux d'évaluation, et les composants en gris clair représentent les différentes instances du modèle : initial et enrichi. Les lignes en pointillés représentent les chemins prix en cas de résultat négatif de l'évaluation. Dans le cas où ces évaluations sont positives, le processus est représenté par les lignes continues. Comme le modèle le suggère, celui-ci peut être évalué de trois manières différentes, dont deux sont situées dans la même branche. Nous voyons également comment le résultat de l'évaluation détermine le déroulement de l'optimisation. Le détail de cette implémentation est présenté dans la seconde partie de notre mémoire.

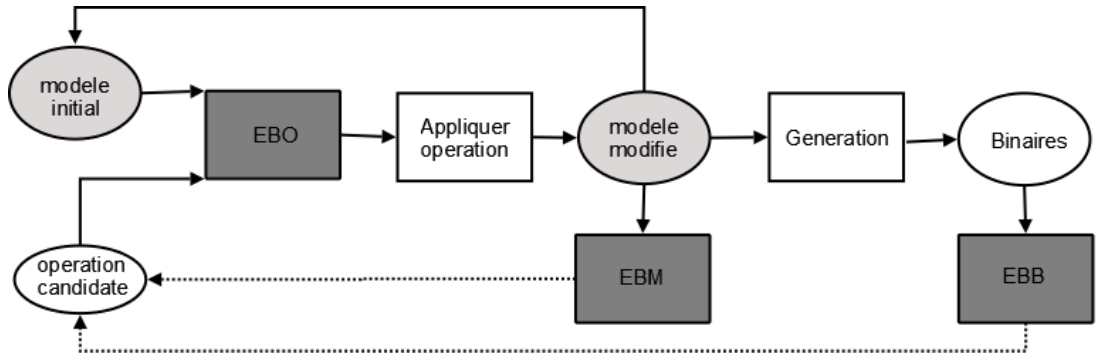


FIG. 4.1 – Pipeline d'évaluation

4.5 Conclusion

Dans le chapitre 1, après avoir présenté la motivation de nos travaux, nous avons présenté le contexte dans lequel se déroulait notre étude, celui des systèmes temps-réel embarqués critiques, ainsi que celui du cycle de développement de ses systèmes, dans lequel nous avons inséré notre processus. Après avoir présenté rapidement notre cas d'étude, nous avons défini et motivé les contraintes que nous voulions respecter pour l'optimisation, puis les problématiques qui en découlaient. Dans le chapitre 2, nous avons montré la cohérence et l'intérêt d'une approche de l'optimisation basée sur les modèles, puis établi une étude comparative de quelques formalismes pour la modélisation des sytèmes TRE, ainsi que des approches d'optimisation et de vérification de contraintes. Finalement, nous utilisons cette étude pour sélectionner les formalismes et techniques dont nous nous servons (AADL et génération de code) et ceux que nous avions à concevoir (langage de description de contrainte, opérations d'optimisation et techniques de validation) dans le cadre de notre thèse. Dans le chapitre 3, nous avons exposé les opérations d'optimisation possibles au niveau du code source, puis les opérations sur les modèles *Merge* et *Move*, qui mettent en œuvre ces optimisations. Dans le chapitre 4, nous établissons une classification du problème du pilotage des opérations *Merge* et *Move*, puis proposons plusieurs solutions algorithmiques pour le résoudre. Nous discutons ensuite de leurs intérêt respectifs, et en déduisons notre choix : une heuristique gloutonne inspirées des solutions classiques pour le problème du sac à dos quadratique. Nous proposons pour cela deux algorithmes distincts, *FGH* et *HGH*. Ces algorithmes nécessitant une évaluation du système, nous discutons ensuite des techniques existantes pour l'évaluation des systèmes TRE. Nous proposons ensuite des solutions pour adapter ces évaluations au niveau des modèles, puis livrons les heuristiques que nous avons choisies. Finalement, nous montrons comment insérer cette évaluation dans le processus d'optimisation.

Dans la seconde partie de ce mémoire, nous présentons la mise en œuvre des propositions faites dans les sections qui précèdent. Dans le chapitre 5, nous présentons notre langage d'expression de contraintes pour AADL, REAL, ainsi que des applications de celui-ci pour la validation de modèles optimisés ou pour rendre plus sûre la réutilisation de composants REAL. Dans le chapitre 6, nous présentons l'implantation du processus d'optimisation et de validation au sein de l'outil Ocarina, puis expliquons comment utiliser les informations contenues dans le modèle pour automatiser l'évaluation des binaires (par la création d'un *modèle enrichi*, ou pour assurer la conservation des propriétés structurelles des modèles au cours des différentes phases d'optimisation. Finalement, nous présentons dans le chapitre 7 une version détaillée de l'implantation de notre cas d'étude, et expliquons les résultats obtenus avec les différentes heuristiques propo-

sées dans la première partie de ce mémoire. Nous concluons dans le chapitre 8 en résumant les contributions que nous avons apportées au problème de l'optimisation dirigée par les modèles dans les systèmes TRE, avant de citer les principales améliorations et apports qui pourraient être apportés à notre travail.

Deuxième partie

Mise en Œuvre

5

Vérification des contraintes sur les modèles

5.1 Introduction

A cause de leur grande diversité, les systèmes embarqués temps-réel doivent pouvoir vérifier un très large spectre de contraintes non-fonctionnelles. Certaines de ces contraintes sont généralisables : strict déterminisme temporel, adéquation logiciel/matériel et sureté d'exécution. D'autres contraintes, au contraire, sont liées au domaine ou au contexte, au nombre desquelles on peut citer le niveau de consommation énergétique ou le débit d'information.

La capacité à vérifier ces contraintes est donc un verrou majeur pour le développement des systèmes embarqués temps-réel. Leur diversité rend nécessaire un langage spécifique qui permette de les exprimer. Dans le cadre d'une approche dirigée par les modèles, ce langage doit pouvoir exprimer les contraintes sur les modèles, de façon à exclure les modèles invalides à un stade précoce du développement, diminuant ainsi le coût des erreurs.

Dans ce chapitre, nous présentons `Requierement Enforcement and Analysis Language` (REAL), un langage permettant de vérifier des contraintes et de calculer des expressions complexes sur les modèles architecturaux AADL. Nous présentons également plusieurs applications de ce langage, d'une part pour vérifier la cohérence du dimensionnement de l'application (projet MOSIC), puis pour la vérification de la sécurité des communications au sein de la plateforme POK.

5.2 Structure de REAL

REAL reprend les concepts des langages d'interrogation et de recherche largement utilisés dans les bases de données (SQL...). Empruntant la plupart de ses notions à la théorie des ensembles et à la grammaire AADL, il est d'accès aisé à tout développeur ayant des notions élémentaires de mathématiques.

L'élément de base du langage est le *theorem*, qui est la plus petite unité logicielle qui puisse être exécutée indépendamment.

5.2.1 Types de bases et ensembles pré-définis

Pour pouvoir interroger le modèle sur lequel les contraintes sont définies, REAL utilise des ensembles pré-définis correspondant aux catégories de composant AADL : `thread`, `process`,

`data`, `bus`, etc. Certains éléments AADL qui ne sont pas des composants sont également présents dans les ensembles, tels que les flots d'exécution et les connexions. Ces ensembles pré-définis contiendront tous les composants de la catégorie correspondante instanciée dans le modèle AADL. Le paramétrage du théorème permet également de définir des ensembles ne contenant que les instances d'un composant particulier (cf. 5.2.5).

Si certaines requètes portent sur les ensembles eux-même (c'est-à-dire sur leur cardinalité), la plupart vont vérifier des propriétés sur leurs ensembles, éventuellement en les associant dans des fonctions de type somme, maximum, produit, etc. Pour pouvoir accéder aux propriétés des éléments AADL, on peut appliquer la fonction `get_property_value` sur l'ensemble ciblé, qui va retourner la valeur de chaque élément de l'ensemble sous la forme d'une liste.

Un autre ensemble particulier est l'ensemble `local_set`. Son contenu varie selon la façon dont le théorème REAL a été appelé (cf. 5.3). Son contenu est nommé le *domaine* du théorème.

Au nombre des valeurs qui peuvent être supportées (donc retournées par `get_property_value`), nous avons tous les types de bases (booléens, entiers, réels, chaines de caractère), mais également les couples et les listes. Les autres types valides dans la seconde version de AADL, tels que les tableaux et les records, ne sont pas supportés dans la version actuelle de REAL.

5.2.2 Theorem et définition de la portée

Le `theorem` doit être vérifié sur tous les élements d'un ensemble particulier, nommé *range set*. La *verification expression* va spécifier une formule qui doit être vérifiée pour l'élément courant du *range set*, accessible à chaque itération grâce à la *range variable*, associée au *range set* lors de sa déclaration.

```
theorem rma_schedulability

  foreach p in Processor_Set do
```

Listing 5.1 – Déclaration d'ensemble de portée

Dans le listing 5.1, la `range variable` est nommé p, et le `range set` qui lui est lié vaut `Processor_Set`, lequel est l'ensemble canonique contenant toutes les instances de processeurs présentes dans le modèle AADL.

5.2.3 Construction des ensembles et variables

Ensembles intermédiaires

L'ensemble des ensembles accessibles par le `range set` et les ensembles pré-définis sont limités. Pour pouvoir décrire des ensembles plus fins, REAL permet de définir des ensembles intermédiaires, formés à partir d'un sur-ensemble de façon similaire aux jointures SQL : dans l'ensemble intermédiaire final se trouveront tous les éléments des sur-ensembles vérifiant une expression donnée. Cette expression peut contenir les mêmes fonctions élémentaires que l'expression de vérification.

Formellement, une définition d'ensemble est décrite par :
$S := \{x \ in \ E \mid f(x)\}$, où x est la variable associée successivement à chaque élément du sur-ensemble E, lui-même défini par une expression sur les ensembles, et pouvant faire référence à tous les ensembles définis précédement dans le théorème ainsi qu'aux ensembles pré-définis. '|'

est l'abbréviation pour *tel que*, et $f(x)$ une expression à valeur booléenne. x sera compris dans l'ensemble S si et seulement si f est vraie pour la valeur x.

L'expression f peut contenir un type particulier de fonction, nommées *fonctions prédicats*, qui servent à exprimer une relation hiérarchique entre deux éléments du modèle AADL, comme par exemple la relation *A est sous-composant de B* ou encore *A est connecté à B*. Les fonctions prédicats permettent de retrouver les informations contenues dans la hiérarchisation du modèle.

```
theorem rma_schedulability

  foreach p in Processor_Set do

    S1 :=  {pr in Process_Set | (is_bound_to (pr, p))};

    S2 :=  {th in Thread_Set | (is_subcomponent_of (th, S1))};
```

Listing 5.2 – Déclaration d'un ensemble intermédiaire

Dans le listing 5.2, nous définissons un ensemble nommé $S1$, défini comme contenant les éléments du sur-ensemble formé par `Process_Set` qui vérifient le prédicat `is_bound_to` avec l'élément de portée courant. L'ensemble $S1$ contiendra donc après construction les instances de processus référencés par la propriété standard AADL `actual_processor_binding` dont la valeur sera l'élément courant de *l'ensemble de portée*. Dit plus simplement, tous les processus s'éxécutant sur ce processeur.

Nous définissons également un second ensemble $S2$ qui est défini sur le sur-ensemble `Thread_Set`, comme contenant les éléments vérifiant le prédicat `is_subcomponent_of` avec l'ensemble $S1$ — c'est-à-dire l'ensemble des instances de threads étant un sous-composant de au moins une des instances de processus présentes dans $S1$.

Définitions de variables

Pour simplifier les expressions et limiter le nombre de calculs, il est possible de définir des variables (d'un type supporté par REAL) de la même manière que l'on définit des ensembles.

Il existe deux manières de définir des variables : soit en leur associant une expression (dont le type de retour va définir le type de la variable), soit en leur associant un sous-théorème (cf. 5.2.5).

La portée d'une variable, comme d'un ensemble intermédiaire, est normalement limitée au théorème courant, à l'exclusion de la partie précédant sa déclaration. Cependant, une variable peut être déclarée comme `global`, auquel cas sa portée s'étend aux sous-théorèmes appelés par le théorème courant.

```
theorem rma_schedulability

  foreach p in Processor_Set do

    var str_periodic := "periodic";

    global var max_same_period := 1;
```

Listing 5.3 – Déclaration d'une variable

Dans le listing 5.3, nous définissons une variable *str_periodic* qui contient une chaine de caractères égale à "periodic". Nous définissons ensuite une variable globale *max_period_equals*, à valeur (implicitement) entière, et égale à 1. Dans ce dernier cas, la variable pourra être référencée dans les sous-théorèmes appelés par le théorème *rma_schedulability*.

5.2.4 Expressions paramétrées

Les expressions paramétrées permettent de définir une expression à appliquer à tous les éléments d'un ensemble. Une expression paramétrés va donc indiquer un élément, une variable locale qui contiendra l'élément courant de l'ensemble, et une expression. Cette dernière peut elle-même contenir une expression paramétrée. Selon le type de retour de l'expression, l'expression paramétrée retournera une liste de ce type. Le listing 5.4, qui calcule le maximum des périodes des threads de chaque process multiplié par 2 et divisé par 500, et de leur propriété *compute_execution_time*, est un exemple d'une telle expression. On remarquera que, dans l'exemple donné, l'expression la plus imbriquée ne dépend pas de l'expression qui la contient. Il est possible que ce soit le cas.

```
theorem variables_imbricated_iterative_expressions

foreach e in Processor_Set do

  Proc_Set(e) := {x in Process_Set | Is_Bound_To (x, e)};

  Threads := {x in Thread_Set | Is_Subcomponent_Of (x, Proc_Set)};

  var y := Max (Expr (Threads, t,
                  ((2 * Max (Expr (Threads, p,
                       (Get_Property_Value
                         (p, "Period")))) / 500)
                     + Last
                       (Get_Property_Value
                         (t, "Compute_Execution_Time")))));

  check (y = 7.0);
end variables_imbricated_iterative_expressions;
```

Listing 5.4 – Expressions paramétrées imbriquées

5.2.5 Appel de sous-théorèmes

Sous-théorèmes et variables

Le calcul de certaines variables ne peut parfois pas être fait avec une simple expression, parce qu'il fait référence à des éléments qui ne peuvent, par exemple, être atteints par les ensembles intermédiaires du théorème courant. Dans ce cas, il est nécessaire d'appeler un sous-théorème grâce au mot-clef `compute`. Ce théorème peut être paramétré par :

- le *domaine* ;
- jusqu'à 10 variables prédéfinies (de argv0 à argv9).

Si un domaine n'est pas spécifié, le sous-théorème héritera du domaine du théorème appelant.

```
theorem rma_schedulability

  foreach p in Processor_Set do

    S1 :=  {pr in Process_Set | (is_bound_to (pr, p))};

    S2 :=  {th in Thread_Set | (is_subcomponent_of (th, S1)};

    S2P :=  {th in S1 | (property (th, "dispatch_protocol") = "periodic")};

    var threads_with_same_period := compute same_period (S2P);
```

Listing 5.5 – Déclaration d'une variable avec sous-théorème

Dans le listing 5.5, nous définissons une variable *threads_with_same_period* grâce à l'exécution d'un sous-théorème *same_period* (défini en 5.7). Nous lui assignons un domaine spécifique, ici égal à $S2P$, un sous-ensemble de $S1$ (cf 5.2) contenant uniquement les threads périodiques. Dans le cas présent, nous n'avons pas besoin d'assigner de variables en paramètre.

Sous-théorèmes requis

Un théorème REAL peut devoir vérifier la validité de certaines contraintes pour pouvoir lui-même être vérifié. Cette approche permet de rendre les expressions modulaires et facilement réutilisables. Il est possible dans REAL de spécifier une liste de théorèmes (nécessairement booléens) à vérifier, grâce au mot-clef `require`. Dans le cas des théorèmes adressés par ce mot-clef, il n'est pas possible pour la version actuelle de REAL d'assigner des paramètres.

5.2.6 Expression de vérification

L'expression de vérification teste une expression sur les ensembles et variables définies dans le théorème, et ce pour chaque *range element*. Cette expression peut concerner soit les ensembles eux-mêmes, soit leurs éléments, soit les différentes valeurs définies dans les variables ou calculées à partir d'expressions élémentaires.

On distingue essentiellement trois types de fonctions dans l'expression de vérification :

- les fonctions ensemblistes, s'appliquant à des ensembles ou des éléments (eg. `cardinal`, `get_property_value`, `is_property_exists`, etc.) ;
- les fonctions élémentaires, s'appliquant sur des types élémentaires (listes y compris) (eg. `sum`, `max`, `size`, etc.) ;
- les fonctions d'agrégat, s'appliquant sur l'ensemble des valeurs calculées pour tous les éléments du *range set* (eg. `mmax`, `msum`).

Les fonctions d'agrégats permettent de calculer une valeur sur tous les éléments d'un ensemble, et sont obligatoirement utilisées pour obtenir des résultats non-booléens à un théorème. Typiquement, un théorème appelé grâce au mot-clef `compute` d'une déclaration de variable devra comporter une fonction d'agrégat, tandis qu'un sous-théorème appelé avec `require` n'en comportera pas.

```
theorem rma_schedulability
  foreach p in Processor_Set do
    S1 :=  {pr in Process_Set | (is_bound_to (pr, p))};
    S2 :=  {th in Thread_Set | (is_subcomponent_of (th, S1)};
    var str_periodic := "periodic";
    global var max_same_period := 1;
    S2P :=  {th in S2 | (property (th, "dispatch_protocol") = str_periodic)};
    S2S :=  {th in S2 | (property (th, "dispatch_protocol") = "sporadic")};
    var threads_with_same_period := compute same_period (S2P);
  check ((property (p, "scheduling_protocol") /= "RMA") or
```

```
        ((threads_with_same_period <= max_same_period) and
         ((Cardinal (S2P) + Cardinal (S2S)) = Cardinal (S2))));

end rma_schedulability;
```

Listing 5.6 – Expression de vérification

Le listing 5.6 vérifie pour chaque élément de portée (ie. pour chaque processeur instancié) que :

- si le protocole d'ordonnancement est *RMA* alors :
- chaque thread périodique a une période unique ;
- un thread est soit périodique, soit sporadique.

Ce théorème terminant par `check`, il ne retournera qu'une valeur booléenne.

Pour mettre en œuvre ce théorème, nous avons fait appel au sous-théorème présenté dans le listing 5.7, et dont l'appel a déjà été présenté en 5.5. On notera que ce sous-théorème utilise son domaine (`Local_Set`) comme ensemble de portée, et pour chaque élément, cherche les threads du domaine ayant la même période. Le théorème retournant une valeur (entière), nous avons dû utiliser la fonction d'agrégat `Mmax`, qui va retourner la valeur maximale des valeurs calculées par l'expression de vérification pour chaque valeur successive de la variable de portée.

```
theorem same_period

  foreach e in Local_Set do

    var period := property (e, "period");

    S := {t in Local_Set | (property (t, "period") = period)};

  return (mmax (cardinal (S)));

end same_period;
```

Listing 5.7 – Théorème avec retour

5.3 Utilisation de REAL

Il existe deux façons de définir un théorème REAL, lesquelles déterminent le type de paramétrage possible : incluse dans une annexe REAL, ou à travers une bibliothèque de théorèmes.

5.3.1 Via une annexe AADL

REAL a été intégré comme annexe du langage AADL, et en tant que tel peut être associé à n'importe quel composant AADL. L'outil Ocarina permet d'évaluer automatiquement la valeur des théorèmes de chaque composant. Dans ce cas, le domaine d'un théorème est composé de toutes les instances du composant propriétaire de l'annexe dans lequel il est décrit.

```
thread a_thread
features
  msg_in : in event port;

annex real_specification {**

  theorem non_empty_thread
```

```
    foreach set in local_set do
      callee := {spg in subprogram_set | is_called_by (spg, set)};
    check (callee > 0);
  end non_empty_thread;
  **};
end a_thread;
```

Listing 5.8 – Appel d'un théorème en annexe

Le listing 5.8 illustre un théorème *non_empty_thread*, ajouté en annexe d'un thread AADL, et vérifiant que ses instances appellent au moins un sous-programme. Un tel mécanisme permet d'interdire une instanciation directe du composant, mais oblige à passer par des implémentations.

5.3.2 Via une bibliothèque de théorèmes

Il est possible de définir des théorèmes REAL dans un fichier à part, qui sert alors de bibliothèque de théorèmes. Dans ce cas, aucun domaine ne peut être défini a priori et il est donc important de vérifier le contexte d'exécution, par exemple au moyen du mot-clef `require`.

```
theorem test_non_empty_thread
  foreach set in local_set do
    callee := {spg in subprogram_set | is_called_by (spg, set)};
    require (is_thread);
  return (mmin (cardinal (callee)));
end test_non_empty_thread;
theorem is_thread
  foreach set in local_set do
    self := {thr in thread_set | (thr = set)};
  check (cardinal (self) = 1);
end is_thread;
```

Listing 5.9 – Spécification d'un théorème en dans une bibliothèque

Le listing 5.9 illustre deux théorèmes fournis par une bibliothèque REAL. Le premier théorème calcule le minimum des cardinaux des sous-programmes appelés par le thread appelant. Il doit vérifier son contexte d'exécution puisqu'il ne sait pas où il va être appelé. Ici, il appelle *is_thread*, qui vérifie que le domaine contient bien des instances de thread. Il peut ensuite être appelé, soit directement via la ligne de commande, soit en annexe avec les mots-clef `compute` ou `require`. Dans ce cas, on peut réécrire le modèle AADL 5.8 comme dans le listing 5.10.

```
thread a_thread
features
  msg_in : in event port;

annexe real_specification {**
```

```
  theorem non_empty_thread

    foreach set in local_set do

      var called_spg := compute test_non_empty_thread (set);

    check (cardinal (called_spg) > 0);

  end non_empty_thread;

  **};
end a_thread;
```

Listing 5.10 – Appel d'un théorème de bibliothèque en annexe

5.3.3 Appels de théorèmes

Les théorèmes en annexe sont appelés automatiquement lors de l'analyse du modèle AADL. Ils peuvent cependant également être appelés isolément, tout comme les théorèmes définis en bibliothèque :

- soit dans le cas de sous-théorèmes (par les mots-clefs `require` et `compute`) ;
- soit dans le cas d'appels de primitives, que ce soit directement dans le code-source ou en utilisant la ligne de commande.

Dans le cas de théorèmes appelés par une primitive dans le code-source, il est possible de construire le domaine manuellement, plutôt que de désigner un ensemble construit par un théorème REAL. Cette application permet une grande souplesse d'utilisation.

5.4 Applications de REAL

5.4.1 Vérification de dimensionnement avec REAL

AADL permet de définir dans le même formalisme la partie logiciel du système TRE, et l'architecture matérielle sur lequel il sera déployé. Néanmoins, pour que ce déploiement soit possible, il faut vérifier qu'un certains nombre de contraintes de dimensionnement sont respectées, c'est-à-dire que l'architecture matérielle fournit des ressources suffisantes pour le fonctionnement du système logiciel. L'une des premières applications de REAL fut de formuler un certain nombre de ces contraintes. Les résultats de ces travaux ont été exploités par le projet ANR MOSIC, ainsi que dans le cadre de l'objectif principal de notre thèse.

Mémoire suffisante

Savoir si une architecture fournit assez de mémoire pour chaque processus d'un système est un problème relativement simple à résoudre, puisqu'il s'agit de sommer quatre paramètres d'un programme pouvant nécessiter une allocation de mémoire :

- le calcul de la taille de la pile pour chaque thread du process doit être résolu. Le chapitre 4 présente une technique se basant sur des outils externes de résolution statique. Dans le cadre des travaux présents, nous nous contenterons de considérer que cette valeur a déjà été obtenue par une méthode quelconque, et enregistrée dans le thread correspondant grâce à la propriété du standard AADL `stack_size` ;

- la taille du tas. Dans le cadre des systèmes TRE, nous imposons également un certain nombre de restrictions qui interdisent l'allocation de mémoire dynamique - et qui nous dispense de faire entrer le tas dans le calcul de la mémoire nécessaire ;
- la taille des instructions et des données du programme correspondant peut être soit évaluée statiquement, soit inférée à partir du modèle et des informations pré-existantes dans le modèle (puisqu'il s'agit alors de code-métier, antérieur au développement du système et donc connu).

En AADL, une mémoire est associée à un process avec la propriété `Memory_Binding_Processor`. Le prédicat REAL `is_bound_to` permet de vérifier qu'une telle propriété associe deux instances de composants. Nous pouvons donc en REAL explorer le modèle à partir d'un processus pour y retrouver toutes les mémoires auxquelles il est associé.

Une première version, ne distinguant pas les types de mémoire (read-only, read-write...) est présentée dans le théorème 5.11. Des raffinements sont bien sûr possibles, tel que distinguer les types de mémoires et les bornes passantes en débit pour un placement optimal.

```
theorem Memory_Size

foreach e in Processor_Set do

  Proc_Set(e) := {x in Process_Set | Is_Bound_To (x, e)};

  Threads := {x in Thread_Set | Is_Subcomponent_Of (x, Proc_Set)};

  Mem_Set(e) := {x in Memory_Set | Is_Bound_To (x, e)};

  check (Sum (Get_Property_Value
              (Mem_Set, "RTOS_Properties::Memory_Size")) >
         Sum (Get_Property_Value (Threads, "source_stack_size")));

end Memory_Size;
```

Listing 5.11 – Mémoire suffisante

Processeur suffisant

La question de l'ordonnançabilité d'un système est largement débattue dans la littérature, donnant lieu à des réponses souvent complexes en temps de calcul. Dans le cadre de nos travaux, nous avons conçu un théorème REAL permettant de vérifier la non-ordonnançabilité, rapide à exécuter et donc interessant pour éliminer a priori des solutions perdantes à un stade précoce de l'évaluation.

Trivialement, une condition de non-ordonnançabilité est que la somme des pires temps d'exécution des threads - y compris sporadiques - liés à un processus sur l'hyperpériode des threads, en termes de nombres de cycles, soit supérieure au nombre de cycles fourni par le processeur sur cet intervalle.

De façon analogue aux mémoires, AADL lie un processus et son processeur par une propriété `Actual_Processor_Binding`. Le prédicat REAL `is_bound_to` permet de vérifier que cette propriété associe deux instances de composants et donc, à partir d'un processus, de retrouver le processeur associé. Le théorème 5.12 permet de s'assurer qu'un processus ne vérifie pas les prérequis les plus élémentaires pour l'ordonnançabilité. Notons que nous considérons que les propriétés `Compute_Execution_cycles` et `Processor_Speed_cycles` contiennent respectivement le nombre de cycles maximum consommés durant une itération d'un thread et la vitesse du processeur en cycles par secondes. Une nouvelle version du théorème utilisant des sous-programmes pour

calculer ces valeurs pourrait utiliser les propriétés standards AADL `Compute_Execution_Time` et `Processor_Speed`.

```
theorem scaling_non_schedulability

  foreach cpu in Processor_Set do

    proc_set := {p in Process_Set | is_bound_to (p, cpu)};

    thr_set := {t in Thread_Set | is_subcomponent_of (t, proc_set)};

    var quantum := gcd (property (thr_set, "Period"));

  check (sum (property (thr_set, "Compute_Execution_Time_cycles") =>
         (quantum * property (cpu, "Processor_Speed_cycles")));

end scaling_non_schedulability;
```

Listing 5.12 – Test de non-ordonnançabilité

5.4.2 Vérification de sécurité avec REAL

Le standard ARINC[48], utilisé essentiellement dans le domaine de l'avionique, décrit une interface entre le système d'exploitation et le code applicatif. Outre les problématiques courantes des systèmes embarqués temps-réel (déterminisme, dimensionnement, etc.), son principal apport est l'introduction de concepts rigoureux de partionnement pour les différentes tâches du système. L'isolation temporelle et spatiale des tâches se fait au moyen de *partitions*, auxquelles sont attribué d'un intervalle temporel et un espace mémoire unique pour exécuter leur code. Plusieurs sous-systèmes peuvent donc partager des ressources matérielles tout en asssurant leur isolation réciproque. Dans [32], l'auteur développe un plaidoyer pour généraliser cette approche aux systèmes temps-réel.

Grâce à la suite POK et au langage AADL, nous pouvons spécifier et générer des systèmes respectant le standard ARINC653 [40] Nous devons, cependant, vérifier que les spécifications fournies par le modeleur système respectent bien tous les points évoqués par le standard, quel que soit le module envisagé. En particulier, nous devons vérifier :

- les contaminations possibles entre les modules ;
- les conditions d'accès au processeur ;
- les conditions d'accès à la mémoire physique.

Pour pouvoir vérifier avec REAL ces propriétés, nous nous appuyons sur la correspondance ARINC653/AADL définie dans le cadre des travaux sur POK [38] :

- un `process` ARINC653 devient un `thread` AADL ;
- un `module` ARINC653 devient un `processor` AADL ;
- une `partition` ARINC653 devient un triplet `process`, `virtual processor` et `memory` AADL.

Grâce à ces éléments, nous avons réalisé des bibliothèques de théorèmes REAL permettant de vérifier ces conditions sur les modèles AADL. Ces théorèmes sont intégrés dans la suite POK [38], et permettent d'accélérer considérablement le cycle de développement en détectant automatiquement les architectures rompant les contraintes de sécurité à un stade précoce de la conception. Dans les parties suivantes, nous présentons chacune de ces contraintes, et donnons des exemples de théorèmes REAL les implantant.

Non-contamination des modules

Un grand nombre de travaux ont été effectués sur les problématiques de contamination dans les systèmes partitionnés. L'approche MILS [28] reprend et synthétise ces travaux. Il s'agit d'assurer qu'un module de haute criticité ne puisse pas être contaminé par un module de plus basse criticité, et que les niveaux de sécurité sont cohérents.

La vérification de Bell-Lapadula 5.4.1 est un exemple de contraintes que doivent vérifier les systèmes [22]. Pour vérifier cette contrainte, nous avons développé le théorème 5.13.

ARINC653 5.4.1 (Bell-Lapadula) *Le niveau de sécurité d'une partition émettrice doit être supérieur ou égal au niveau de sécurité des partitions réceptrices connectées à celle-ci.*

```
theorem bell_lapadula

    foreach p_src in process_set do

      VP1 := {x in Virtual_Processor_Set | is_bound_to (p_src, x)};

      B_Src := {x in Virtual_Bus_Set | is_provided_class (VP1, x)};

      P_Dest := {x in Process_Set | Is_Connected_To (p_src, x)};

      VP2 := {x in Virtual_Processor_Set | is_bound_to (P_Dest, x)};

      B_Dst := {x in Virtual_Bus_Set | is_provided_class (VP2, x)};

    check (Cardinal (P_Dest) = 0 or
            (max (property (B_Src, "POK::Security_Level")) <=
             min (property (B_Dst, "POK::Security_Level"))));

end bell_lapadula;
```

Listing 5.13 – ARINC653 : Bell-Lapadula

La vérification MILS 5.4.2 est nécessaire pour assurer la cohérence des niveaux de sécurité [82]. Cette contrainte est vérifiée à l'aide du théorème 5.14.

ARINC653 5.4.2 (MILS) *Si des partitions sont connectées, leur niveau de sécurité doivent être identique.*

```
theorem MILS

    foreach p_src in process_set do

      VP1 := {x in Virtual_Processor_Set | is_bound_to (p_src, x)};

      B_Src := {x in Virtual_Bus_Set | is_provided_class (VP1, x)};

      P_Dest := {x in Process_Set | Is_Connected_To (p_src, x)};

      VP2 := {x in Virtual_Processor_Set | is_bound_to (P_Dest, x)};

      B_Dst := {x in Virtual_Bus_Set | is_provided_class (VP2, x)};

    check (Cardinal (P_Dest) = 0 or
            ((max (property (B_Src, "POK::Security_Level")) =
              max (property (B_Dst, "POK::Security_Level"))) and
             (min (property (B_Src, "POK::Security_Level")) =
              min (property (B_Dst, "POK::Security_Level")))));
```

```
end MILS;
```

Listing 5.14 – ARINC653 : MILS

Accès au processeur

Typiquement, il s'agit de vérifier que l'allocation des intervalles temporels d'exécution pour chaque partition correspond bien à des ressources existantes. En d'autres termes, il s'agit d'un vérificateur d'ordonnancement très simple. La vérification 5.4.3 exprime cette contrainte que nous implémentons avec le théorème 5.15. D'autres vérifications sont bien sûr possibles, telles que savoir si chaque intervalle est bien assigné à une unique partition.

ARINC653 5.4.3 (Processor Access) *Pour chaque module, la somme des intervalles de temps assignés aux partitions est inférieure ou égale au temps dont dispose le module durant une période.*

```
theorem POK_scheduling

    foreach cpu in processor_set do

    check (float (property (cpu, "POK::Major_Frame")) <=
           float (sum (property (cpu, "POK::Slots"))));

end POK_scheduling;
```

Listing 5.15 – ARINC653 : Processor Access

Accès à la mémoire

Les conditions de sûreté pour les accès aux composants physiques, en particulier les mémoires, ont été développées par plusieurs études dont [26]. Un exemple de contrainte à vérifier est donné en 5.4.4.

ARINC653 5.4.4 (Memory Access) *L'accès à un composant de mémoire physique n'est autorisé que pour des partitions de même niveau de sécurité.*

```
theorem One_Security_Level_By_Memory

    foreach m in memory_set do

      P := {x in Process_Set | is_bound_to (x, m)};

      VP := {x in Virtual_Processor_Set | is_bound_to (P, x)};

      B := {x in Virtual_Bus_Set | is_provided_class (VP, x)};

    check (all_equals (property (B, "POK::Security_Level")));

end One_Security_Level_By_Memory;
```

Listing 5.16 – ARINC653 : Memory Access

5.4.3 Assurer la sûreté du mécanisme d'extension de ADDL

Réaliser des bibliothèques de composants réutilisables est un enjeu important de l'ingénierie des modèles. L'intérêt d'une telle approche est de permettre non seulement un gain de temps lors de la conception du système, mais également lors de son développement, puisqu'un modèle développé est généralement associé à une implantation existante.

La réutilisation des modèles existants présente cependant des difficultés intrinsèques. Des hypothèses peuvent être faites dans le modèle original (ou, plus fréquemment encore, sur le code-métier associé à ce modèle), et celles-ci doivent être respectées par les modèles héritant du modèle original. Ces hypothèses doivent être exprimées sous forme de contrats présents dans la documentation associée au modèle. Les hypothèses implicites (non contractuelles) peuvent provoquer de graves défaillances du système quand elles sont violées — un exemple en est la destruction du vol inaugural de la fusée Ariane V, causé par une erreur d'intervalle de valeur d'une mesure [49].

AADL définit à la fois une grammaire complète et des règles de légalité et de nommage qui limite l'ensemble des modèles valides. Ces règles définissent des vérifications de cohérence qui assurent que le modèle est pertinent. Elles ne s'appliquent, cependant, qu'aux mécanismes principaux de AADL, et en particulier ne permettent pas d'assurer un contrôle efficace sur les composants qui partagent une relation d'héritage avec composant donné (que ce soit en l'implémentant, en le raffinant ou en l'étendant avec le mot-clef *extend*). Nous définissons trois types d'erreurs possibles :

1. usage incohérent des propriétés : si les propriétés sont typées, aucune vérification n'est effectuée sur l'utilisation coordonnée des propriétés — il est donc possible de définir un entier dont le codeset est cyrilique ;
2. incohérence dans les extensions de composants : il n'est pas possible de limiter l'ajout de features ou de propriétés dans les dérivés d'un composant. C'est pourtant une option importante dans tout système intégrant une notion d'héritage que de pouvoir décrire une impossibilité — comme par exemple l'ajout de nouveaux bus dans un processeur, pour des raisons d'énergie ou de sécurité ;
3. les incohérences introduites par l'usage de langages d'annexes AADL. Il est possible avec ces annexes de spécifier certains aspects du système (gestion des erreurs, comportement, etc.). Cependant, il s'agit de langages spécifiques à AADL avec leur sémantique propre — il est donc difficile de vérifier leur impact sur le système. Nous ne traitons donc pas ce dernier problème.

Dans la suite de cette partie, nous illustrons successivement comment traiter les points 1 et 2 avec REAL.

Vérification de la cohérence des propriétés

Assurer que les différentes propriétés sont cohérentes est trivial dans le cas d'un composant unique, mais devient délicat dans le cas où celui-ci est hérité. En effet, il est peu réaliste et contraire au but de la réutilisation de supposer que l'architecte a connaissance de toutes les propriétés associées aux différents niveaux d'implémentation successifs du composant réutilisé.

Dans AADL, les propriétés sont décrites dans des *property sets*. Certaines propriétés ont, par définition, des relations de causalité entre elles (par exemple, la propriété `real_range` qui définit la portée d'un nombre réel implique que la propriété `data_representation` existe et vaille

`float`. Nous proposons de décrire au niveau de la définition du `property set` les relations de causalité, de sorte que la relation soit vérifiée sur tout modèle possédant la propriété associée.

Ainsi, supposons que nous décrivions en AADL la propriété évoquée précédement, tel que l'illustre le listing 5.17.

```
Real_Range: range of aadlreal
  applies to ( data, port, parameter );

  -- Real_Range definit un intervalle de valeur reelles of real
  -- qui s'applique a un data component. Cette propriete doit
  -- s'appliquer a un type reel
```

Listing 5.17 – Déclaration de la propriété Real_Range

Cette propriété ne devrait s'appliquer qu'aux données de type réel. Le type d'une donnée est spécifié par la propriété standard `data_representation`. Celles-ci sont limitées à *Array*, *Boolean*, *Character*, *Enum*, *Float*, *Fixed*, *Integer*, *String*, *Struct* et *Union*. Parmi ces types, seul *Float* est de type réel.

Nous pouvons donc écrire le théorème REAL 5.18 qui, associé à la propriété `Real_Range`, vérifie que tous les composants de type data munis de cette propriété vérifient également $data_representation = float$.

```
--  Real_Range property

theorem check_real_range
  foreach d in data_set do

  --  1/ Verifier que la propriete "Real_Range" ne soit appliquee
  --     qu'a des type de donnees dont la representation est flottante.

  check ((not property_exists (d, "real_range"))
       or (property_exists (d, "data_representation")
        and get_property_value (d, "data_representation") =
                        "float"));
end check_real_range;
```

Listing 5.18 – Théorème associé à Real_Range

Vérification du contexte d'héritage

L'héritage peut remettre en cause les hypothèses implicites sur lesquelles reposent les composants réutilisés. Il convient donc de les rendre explicites. En les formulant avec REAL, on assure au niveau du modèle original que l'utilisation qui en sera faite aura lieu dans un contexte approprié. La vérification automatisée offerte par l'interpréteur de REAL est à la fois plus sûre et moins chère (en terme d'heures de travail) que la communication par documentation interposée.

Pour pouvoir spécifier un contrat entre un composant et ses dérivés, nous utilisons la règle suivante : un composant dérivé d'un autre hérite des théorèmes REAL spécifiés dans l'annexe de ce dernier. Il suffit donc de décrire le contrat sous la forme d'un théorème REAL en annexe du composant original pour pouvoir vérifier automatiquement le respect du contrat dans tous les composants dérivés.

Le listing 5.22 définit un processeur possédant un accès à un bus en feature. Il est possible d'ajouter de nouvelles features dans les composants étendant ce composant, comme illustré par

le listing 5.20.

```
processor cpu
features
  B : requires bus access C_Bus.Impl;
end cpu;
```

Listing 5.19 – Composant original

```
processor cpu2 extends cpu
features
  B2 : requires bus access C_Bus.Impl;
end cpu2;
```

Listing 5.20 – Composant dérivé

Puisque nous voulons interdire une telle action, nous ajoutons une annexe REAL (listing 5.21) au composant initial, assurant ainsi le respect de cette règle dans les composants étendants (ou, plus généralement, dérivant) de celui-ci. Avec cette annexe, le composant *cpu*2 illustré par 5.20 sera rejeté.

```
theorem unique_bus

  foreach cpu in local_set do

    ports := {p in port_set | is_feature_of (p, cpu)};

    buses := {b in bus_set | is_accessing_to (ports, buses)};

  check (cardinal (buses) = 1);

end unique_bus;
```

Listing 5.21 – Contrat : unicité du bus

Finalement, le listing 5.22 illustre le modèle effectif du composant original *cpu*.

```
processor cpu
features
  B : requires bus access C_Bus.Impl;

annex real_specification {**
  theorem contract_list
    foreach cpu in local_set do

      requires (unique_bus);

    check (1 = 1);
  end contract_list;
**}
end cpu;
```

Listing 5.22 – Composant original contraint

5.5 Rôle de REAL dans le processus d'optimisation

Nous avons vu que nous pouvions spécifier des contraintes et calculer des expressions sur les modèles AADL en utilisant REAL. Si cette possibilité peut être exploitée indépendamment de

tout processus d'optimisation, elle intervient à trois niveaux dans ce dernier :

- la vérification des contraintes exprimées sur le modèle ;
- la vérification de la cohérence entre les données structurelles des différentes versions du modèle ;
- l'évaluation des performances du modèle.

Le premier point est utilisé par l'architecte quand il traduit les contraintes spécifiées au moins pour partie en langage naturel dans le cahier des charges. Ces contraintes doivent être vérifiées à la fois sur le modèle initial et sur les versions optimisées de celui-ci. Avec REAL, il peut non seulement factoriser l'expression de ces contraintes entre les versions des modèles, mais également entre les projets : les contraintes devraient être rédigées avant même le modèle initial.

Le second point permet de vérifier que l'optimisation des modèles ne viole pas certaines de leurs caractéristiques structurelles, qui sont en fait des informations implicites dans le cahier des charges. Nous présentons dans la section 6.3 une technique pour déduire ces caractéristiques du modèle initial, et vérifier les théorèmes associés dans les modèles optimisés.

Le dernier point permet à l'architecte de proposer ses propres critères d'évaluation du système. Nous assurons ainsi avec REAL la flexibilité du processus d'optimisation. La section 7.2 fournit des exemples d'une telle utilisation.

6

Optimisation et évaluation

6.1 Implantation du processus d'optimisation

Nous avons présenté en 4.4 et en 4.2 les contributions concernant l'évaluation et l'optimisation des modèles, puis dans le chapitre 5 les concepts et l'utilisation d'un langage de contraintes. Dans cette section, nous présentons tout d'abord l'architecture de l'outil OCARINA, puis la façon dont nos contributions s'articulent au sein de cette architecture.

6.1.1 L'architecture Ocarina

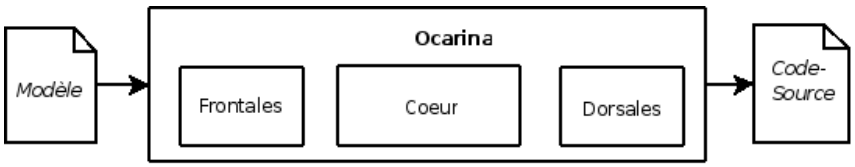


FIG. 6.1 – Ocarina : organisation générale

Le schéma 6.1 illustre l'architecture générale d'OCARINA. L'organisation choisie est proche de celle utilisée par les compilateurs. Les modules de la partie `frontale` contiennent les parsers pour les divers langages en entrée (AADL, mais également les langages d'annexe, officiels ou non, tels que REAL ou l'annexe comportementale). La partie centrale `core` contient les arbres issus des phases de parsing et les éventuels arbres intermédiaires, ainsi que les méthodes permettant d'effectuer les passages de l'un à l'autre, d'analyser leur cohérence, et offre des fonctions de recherche dans ceux-ci. Finalement, les modules de la partie `dorsale` contiennent les arbres représentant les langages en sortie (code-source *Ada* ou *C*, fichiers de compilation, réseaux de Petri...) et les fonctions de transformation des arbres internes vers celui-ci et des arbres de sortie vers le langage-cible.

La partie centrale

La partie centrale d'OCARINA transforme les informations obtenues par les parseurs en deux arbres différents :

– *l'arbre syntaxique*, qui décrit le modèle tel qu'il est défini par l'utilisateur, en résolvant les références ;

– *l'arbre d'instances*, qui étale la structure de l'arbre syntaxique, et créé une *instance* d'un sous-composant pour chaque occurence de son type de conteneur.

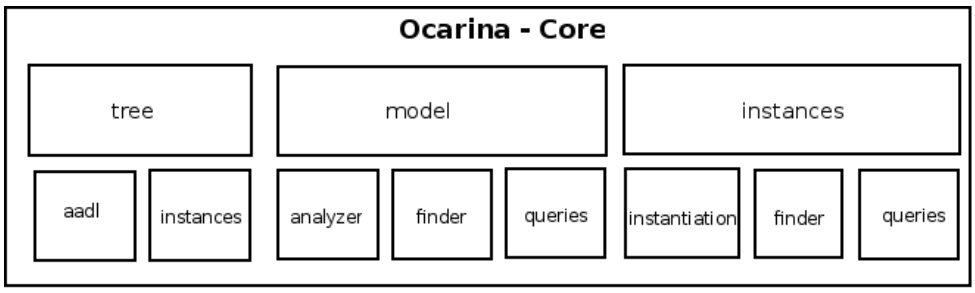


FIG. 6.2 – Ocarina : organisation du coeur

Le schéma 6.2 illustre l'organisation de la partie centrale d'OCARINA. Les deux composants principaux, `model` et `instances` contiennent respectivement les fonctions permettant de construire l'arbre syntaxique et l'arbre d'instances, générés à partir de descriptions de l'arbre en langage de description d'interface (IDL3). Ces descriptions sont définies dans la partie `tree`.

La partie `model` comporte une sous-partie `analyzer`, qui permet de résoudre les références et de vérifier que les règles de légalité définies dans le standard AADL sont respectées. La partie `instances` comporte une sous-partie `instantiation` qui offre des fonctions permettant de produire l'arbre d'instanciation à partir de l'arbre syntaxique. Les deux parties offrent également des fonctions permettant de rechercher un élément particulier dans l'arbre (`finder`) ou d'obtenir la valeur des propriétés associées aux éléments du modèle (`queries`).

Frontales et Dorsales

Les frontales concernent le langage AADL, mais également les langages d'annexes qui lui sont associés, officiellement ou non. Au moment de la rédaction de ces lignes, les annexes suivantes étaient supportées :

– l'annexe comportementale, qui permet de spécifier un comportement au threads et sous-programmes définis dans l'application ;
– REAL, qui permet de définir des contraintes que les différents composants de l'application doivent respecter.

La frontale définit deux opérations principales : le *lexeur*, qui découpe le modèle en entrée en une liste de lexèmes représentant les différents mots-clefs et identifiants composant ce modèle, et le *parseur*, qui construit une instance de *l'arbre syntaxique* à partir de la liste de lexèmes.

Les dorsales quant à elles sont divisées selon le type de sortie sélectionnée :

– les dorsales de code-sources (C, Ada), qui génèrent un système fonctionnel en utilisant un intergiciel (PolyORB-QOS, PolyORB-HI, POK) pour la gestion des communications ;
– la dorsale de preuve (Réseaux de Petri, fichiers TPOF/Assertions de Bound-T), qui génère des descriptions — formelles ou non — de l'application en vue d'analyse par des outils tiers ;
– la dorsale de contraintes (annexe comportementale, REAL), qui teste si le modèle répond aux contraintes définies en entrée.

6.1.2 L'interpréteur REAL

Le lexeur/parseur REAL

Nous avons défini un lexeur et un parseur pour le langage REAL, qui peuvent s'exécuter soit sur des fichiers indépendants, soit sur des annexes AADL. Ces fonctions permettent de construire l'arbre associé au modèle REAL de chacun des théorèmes fournis en entrée. Dans le cas de théorèmes spécifiés dans des annexes AADL, les arbres produits seront des sous-arbres inclus dans l'élément de l'arbre syntaxique AADL correspondant au composant dans lequel le théorème était déclaré. Le schéma 6.3 illustre le positionnement résultant du modèle AADL 6.1.

Dans la figure 6.3, les lignes en pointillés représentent les branches manquantes entre la racine du modèle AADL (`aadl_root`) et le process p, qui contient le thread T. L'ellipse en gris représente la racine de l'arbre REAL, qui est incluse dans le champs `annexes` de T. Les fils de l'arbre REAL ayant pour racine *unique* ne sont pas représentés dans la figure.

```
process p
end p;

process implementation p.impl
  subcomponents
    T1 : thread T;
end p.impl;

thread T
  features
   msg_out : out data port types::integer;

  annexes real_specs {**

    theorem unique

     foreach s in system_set do

     check (sum (cardinal (local_set)) = 1);

    end unique;
  **};

end T;
```

Listing 6.1 – Exemple d'annexe REAL au sein d'un modèle AADL

Analyse et annotation de l'arbre

L'analyse de l'arbre REAL est exécutée en quatre phases :

- *résolution des références* : on retrouve les déclarations correspondant à chaque ensemble et variable, et on les associe dans l'arbre REAL ;
- *résolution du type des propriétés* AADL : pour chaque propriété mentionnée dans le théorème REAL, on cherche si elle est définie dans un `property_set` visible, puis quel est son type ;
- *vérification des règles sémantiques* : pour chaque paramètre des fonctions ou prédicats, on vérifie que le type retourné est bien le type attendu ;
- *analyse des flots* : les flots bout-à-bout du modèle AADL (c'est-à-dire l'ensemble des parcours effectivement possibles dans le graphe des connections) sont calculés et enregistrés.

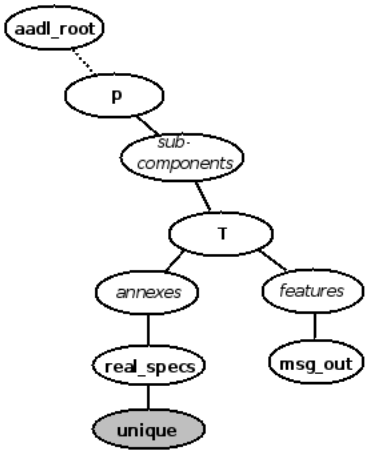


FIG. 6.3 – Inclusion d'un arbre REAL dans un arbre AADL

Ces opérations sont définies dans la partie `model` de la partie centrale d'OCARINA, à l'exception de l'exploration des flots et la résolution du type des propriétés qui nécessitent d'avoir accès aux instances AADL et sont donc définies dans `instances`.

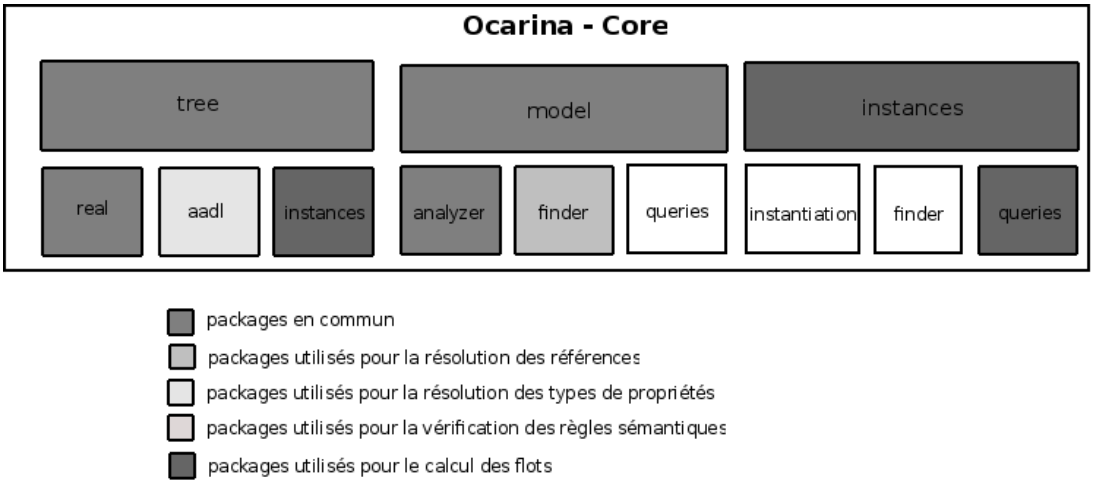


FIG. 6.4 – Dépendances du cœur d'Ocarina pour l'analyse de REAL

La figure 6.4 illustre l'utilisation des packages du cœur d'OCARINA par les différentes phases d'analyse. On notera l'ajout d'un package `real` dans les arbres disponible, qui contient la définition de l'arbre interne associé au langage REAL. On remarquera également que l'analyse de la valeur des propriétés est faite sur l'arbre syntaxique, et non sur l'arbre d'instances. En effet, le type d'une propriété n'est pas instancié (seule sa valeur l'est).

Interprétation et primitives

La dorsale `ocarina.backends.real` permet de vérifier un théorème REAL. L'interprétation contient essentiellement deux problèmes disctincts : la récupération des données de l'arbre

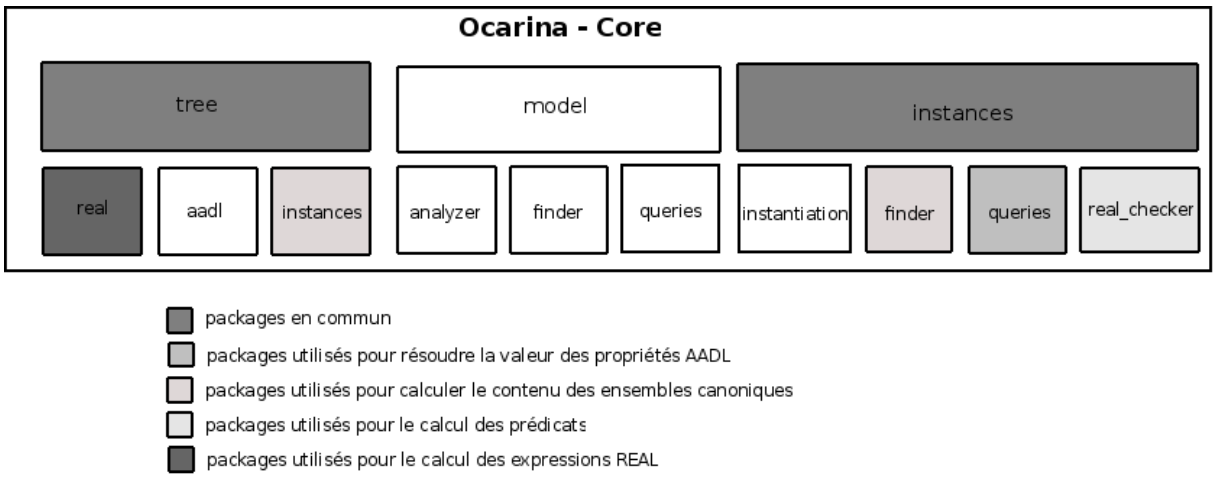


FIG. 6.5 – Dépendances du cœur d'Ocarina pour l'interprétation de REAL

d'instances AADL, et la résolution des expressions définies sur ces données dans le théorème REAL.

Les données de l'arbre d'instances sont nécessaires pour trois points particuliers :
- la valeur des propriétés des composants AADL désignées par le théorème REAL ;
- le contenu des ensembles prédéfinis (`thread_set`, `memory_set`, etc.) ;
- le calcul des prédicats (`is_called_by`, `is_subcomponent_of`, etc.).

Nous offrons dans la sous-partie `instances` diverses méthodes permettant de récupérer des informations de l'arbre d'instances AADL, ou de calculer les prédicats. Le schéma 6.5 illustre ces dépendances. On notera l'ajout du package `real_checker` dans le package `instances`. Il contient les méthodes de calcul des prédicats.

Interprétation d'un théorème REAL

L'interprétation d'un théorème REAL suit les étapes suivantes :

1. construction de *l'ensemble de portée* ;
2. résolution de la valeur liée à *l'élément de portée* courant ;
3. application de la fonction d'agrégat (dans le cas où les mot-clef *return* est utilisé).

La résolution de la valeur liée à l'élément de portée est elle-même divisée en plusieurs phases :

1. construction des ensembles et variables intermédiaires ;
2. résolution de l'expression de vérification.

Construction des ensembles intermédiaires Un ensemble REAL est composé à partir des éléments d'une *expression d'ensemble* vérifiant une expression booléenne. L'interpréteur va effectuer les opération suivantes :

1. créer un ensemble intermédiaire vide ;
2. résolution de l'expression d'ensemble ;
3. pour chaque élément de l'expression d'ensemble, calculer la valeur de l'expression booléenne ;
4. si cette dernière est vraie, ajouter l'élément à l'ensemble intermédiaire.

Construction des variables intermédiaires Les variables intermédiaires peuvent être affectées soit par un expression, soit par l'appel d'un sous-théorème. Dans ce dernier cas, l'appel du sous-théorème est effectué lors de la résolution de la valeur de la variable.

Résolution des expressions Les expressions d'ensemble ou de vérification sont résolues récursivement. Il existe trois types d'expressions : unaires, binaires et ternaires, en référence au nombre d'opérandes associés. Les opérateurs ne supportent généralement qu'un seul type expression, pour lequel leur comportement est défini (dans `ocarina.backends.real`). Les feuilles de l'arbre associé à l'expression sont des litéraux, des variables, des ensembles intermédiaires ou prédéfinis, ou encore des appels de fonctions paramétrés. Dans ce dernier cas, les paramètres sont fortement contraints (la vérification est effectuée par l'analyseur dans la partie statique, et à l'exécution pour la partie dynamique).

Application de la fonction d'agrégat La fonction d'agrégat va permettre de transformer une liste de résultat en résultat unique. C'est une fonction de $\mathbb{R}^k \rightarrow \mathbb{R}$, où k est le nombre d'élément de l'ensemble de portée. Il s'agit typiquement d'opérateurs mathématiques tels que min, max, sum, etc. Contrairement à l'expression de vérification, la fonction d'agrégat ne s'applique que après la résolution de toutes les valeurs associées aux éléments de portées successifs.

6.1.3 Le module Transfo

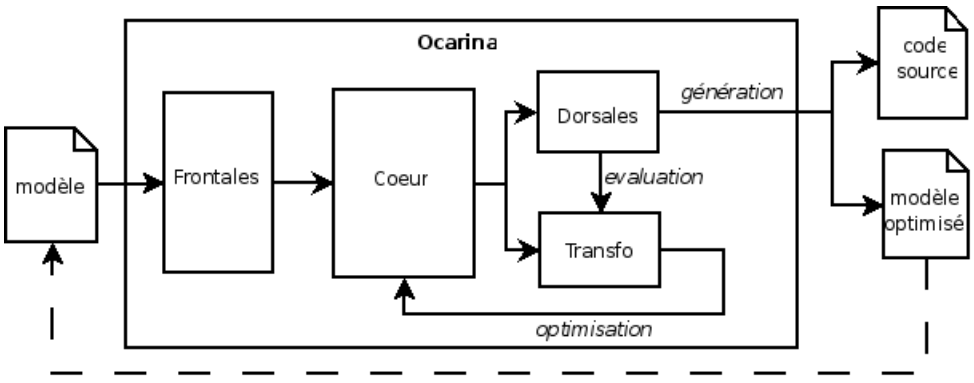


FIG. 6.6 – Positionnement du module Transfo dans Ocarina

Le module `transfo` s'insère dans le cadre entre la partie centrale et les dorsales. Son rôle est de procéder à la transformation du modèle en vue d'obtenir un système réel optimal par rapport aux contraintes définies. Le schéma 6.6 illustre ce positionnement dans l'architecture d'OCARINA.

Package Fusion

Le package *transo.optim.fusion* définit la fonction de `Merge` pour un ensemble de threads AADL. Les détails de l'algorithme sont données en 3.4. La fusion s'applique à l'arbre syntaxique, mais renouvelle en conséquence l'arbre d'instances.

Package Move

Le package *transo.optim.move* définit la fonction de `Move` pour un thread et un process AADL. Les détails de l'algorithme sont données en 3.3. La fusion s'applique à l'arbre syntaxique, mais renouvelle en conséquence l'arbre d'instances.

Package Eval

Le package *transo.optim.eval* définit les opérations d'évaluation pour les fonctions de coût et de valeur des algorithmes d'optimisation définis en 4.2. Pour mener à bien cette évaluation, le package appelle des théorèmes REAL définis dans une bibliothèque qui peut être configurée par l'utilisateur du processus d'optimisation.

Package Optim

Le package *transo.optim* dirige le processus d'optimisation. C'est dans ce package que sont définis les différents algorithmes d'optimisation.

6.2 Construction du modèle enrichi

Nous avons vu en 4.4 que la construction du modèle enrichi nécessite l'extraction d'information obtenue par des outils tiers, à partir du système réel généré pour le modèle courant. Une telle opération fait apparaitre trois problématiques :

- la configuration des outils ;
- l'interprétation des résultats ;
- l'intégration des résultats.

Dans le cadre des évaluations auquelles nous voulons procéder, nous avons conçu un exemple généralisable de ce procédé d'enrichissement du modèle architectural autour de l'outil Bound-T [7] en proposant une solution pour chacune de ces étapes en vue de connaitre le WCET de chaque sous-programme, comme présenté dans l'article [52].

Dans cette section, nous exposons rapidement les points pertinents dans l'utilisation de Bound-T, puis nous présentons notre principe d'enrichissement, avant de présenter individuellement chaque étape.

6.2.1 Analyse du WCET et de la pile avec Bound-T

L'outil Bound-T permet l'analyse de le pile de donnée et du WCET de sous-programmes dans certaines architectures matérielles. Il supporte un mode *Hard Real-Time* qui permet une évaluation déterministe de ces derniers. Pour mettre en place une telle évaluation, il faut configurer trois éléments en entrée.

- la ligne de commande ;
- le fichier *Threads and Protected Objects* (TPO) ;
- le fichier d'assertions.

En fonction de ces entrées et de l'exécutable considéré, un fichier de sortie *Execution Skeleton File* (ESF) est produit. Ce fichier contient les WCETs des sous-programmes définis dans le fichier TPO.

Considérons le code C suivant, calculant la fonction $f \leftarrow \sum_{i=1}^{n} i \sum_{j=1}^{i} j$:

[7] http://www.tidorum.fi/bound-t/

```
int sum (int n){
  int s = 0;
  int i;

  for (i = 0; i < n; i++) s += i;

  return s;
}

int square_sum_job (int n) {
  int x = 0;
  if (n > 10) return -1;

   x = sum (n);
   x = ((x + 1) * x) / 2;

   void_function ();

   return x;
}
```

Listing 6.2 – Code-source de l'application

Nous considererons que ce code se trouve dans un fichier exécutable nommé *square_sum*, compilé pour une architecture sparcv8 à partir du fichier *square_sum.c*. Seul *square_sum* est déclaré dans le header associé au fichier C. Cette dernière est appelée directement par un thread POSIX périodique.

Configuration et fichier d'assertions

Le fichier TPO doit indiquer le sous-programme à analyser. Nous voulons connaitre le WCET total lié à l'opération effectivement appelée par le reste de l'application ; nous devons donc choisir le sous-programme de plus haut niveau. Il s'agira donc de *square_sum*. Le nom du sous-programme à exécuter doit être celui apparaissant dans la table des symboles du binaire compilé : dans le cas de langage définissant des espaces de noms de type `packages` Ada ou java, il sera donc dans la pratique préfixé par le nom de l'espace de noms dans lequel il est déclaré. Dans le fichier TPO, nous indiquerons donc :

```
node square_sum_node

  thread square_sum_thread
    type cyclic;
    root square_sum_job
  end square_sum_thread;

end square_sum_node;
```

Listing 6.3 – Fichier TPO

Le fichier d'assertion permet de spécifier des contraintes sur le code de manière à rendre possible son interprétation dans les cas où Bound-T ne peut pas le faire de lui-même. Dans l'exemple 6.2, nous pouvons observer deux problèmes qui rendent l'analyse impossible par Bound-T :

- la fonction *void_function* est inconnue ;
- la boucle dans la fonction *sum* n'est pas bornée par une constante.

Cependant, il est possible de déduire une borne supérieur sur le nombre d'itération de la boucle de *sum*, puisqu'elle n'est appelée que par *square_sum_job*, et que celle-ci ne peut avoir

de valeur supérieure à 10 pour son paramètre n. Ajouter le code illustré dans le listing 6.4 dans le fichier d'assertions résoudra ce problème :

```
subprogram "sum"
   loop repeats 10 times; end loop;
end "sum";
```

Listing 6.4 – Fichier d'assertions

Supposons que la fonction *void_function* soit connue, et que son temps d'exécution (en nombre de cycles) soit négligeable. On peut alors résoudre le second problème en ajoutant le code illustré par le listing 6.5 dans le fichier d'assertions :

```
subprogram "void_function"
   time 0 cycles;
end "void_function";
```

Listing 6.5 – Fichier d'assertion (2)

La configuration de la ligne de commande dépend essentiellement du binaire à évaluer et de l'architecture pour laquelle il a été compilé. Les options toujours présentes seront -hrt (pour activer l'analyse temps-réel dur), ainsi que les noms du fichier exécutable, du fichier d'assertion et du fichier TPO. Dans le cas que nous avons présenté, celle-ci sera donc : *boundt_sparc −hrt −dev = sparcv8 −assert assertions.txt scenario.tpo square_sum*

Interprétation des résultats

Considérons le fichier ESF 6.6 généré par la commande précédente. Chaque thread spécifié dans le fichier TPO donne lieu à une entrée dans le fichier ESF. Nous observons que trois valeurs différentes ont été calculées à la ligne *wcet*. La première valeur correspond au WCET en terme de cycle processeur, la seconde au pire nombre de lectures en mémoire, et le troisième au pire nombre d'écritures en mémoire. Dans l'exemple que nous avons considéré, le WCET est donc de 378 cycles processeurs par itération du thread.

```
program square_sum
   thread square_sum_thread
     type cyclic
       wcet 378 , 6 , 5
   end square_sum_thread
end square_sum
```

Listing 6.6 – Fichier ESF

6.2.2 Le processus d'enrichissement

Les éléments dont nous avons besoin pour les différents aspects de la configuration de Bound-T peuvent être déduits du modèle architectural. Nous montrons dans la figure 6.7 un processus qui permet de procéder à cette configuration et d'enrichir l'arbre AADL initial avec les résultats de l'analyse. Les parties en gris clair correspondent à nos contributions. Le module d'interprétation de Bound-T permet de déduire la configuration de l'outil à partir du modèle AADL initial, tandis que le module d'interprétation des fichers ESF permet d'enrichir le modèle. Nous présentons ci-dessous précisément chacune des phases de ce processus.

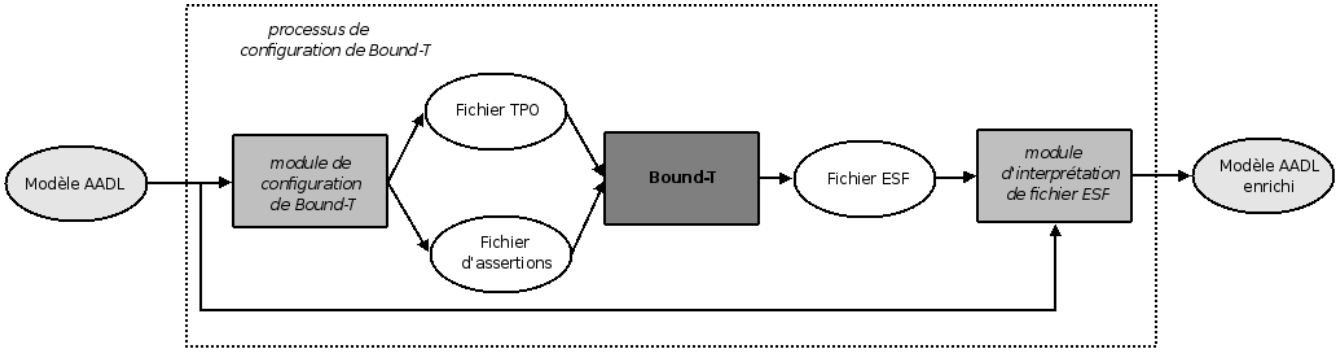


FIG. 6.7 – Bound-T pipeline

fichier TPO/Assertions	modèle AADL
sous-programmes du thread	`threads` calls
Espace de nom du sous-programme	`package`
Nom du programme	`process`
Périodicité	`thread` property *Dispatch_ Protocol*
Architecture cible	`system` property *Deployment_ Execution_ Plateform*
fonction d'envoi	`thread` et `port`

TAB. 6.1 – Table des symboles - AADL

6.2.3 Déduire la configuration du modèle AADL

Construction du fichier TPO

Nous avons vu que la construction du fichier TPO nécessite la connaissance des éléments suivants :

- le nom du programme ;
- le nom et l'espace de nom du sous-programme de plus haut niveau des threads ;
- la périodicité (ie. périodique ou sporadique) des threads.

En fonction de notre connaissance des règles de génération de code et des effets de la compilation, nous pouvons déduire toutes ces informations du modèle AADL, comme indiqué dans le tableau 6.1. Nous sommes donc en mesure de générer un fichier TPO pour chaque process du modèle AADL, qui contiendra pour chaque thread sa périodicité et le sous-programme appelé pertinent.

Construction du fichier d'assertion

La construction du fichier d'assertion repose sur la connaissance des motifs de génération de code. En premier lieu, nous pouvons définir un ensemble de sous-programmes appelés dans le code généré relevant des appels à des fonctions externes, dont nous pouvons trouver la définition dans la documentation du système et/ou de l'architecture. Un exemple d'un tel sous-programme est la division de nombres flottants. Puisque nous savons quelle architecture est notre cible (propriété *Deployment_ Execution_ Plateform* du system contenu dans notre modèle AADL), nous savons également dans quel processeur chercher l'information. Ces informations sont enregistrées dans un fichier par couple architecture/système, qui formera la base de notre fichier d'assertion. Dans les cas où l'information n'a pas pu être trouvée, le WCET de la fonction est considéré comme

nul.

D'autres sous-programmes non bornés viennent de la runtime du code généré à partir du modèle. Ainsi, les fonctions d'envoi et de réception des données sont typiquement non bornées. Une exploration du modèle, cependant, permet de déduire un nombre maximum d'itérations de leurs boucles. Dans le cas de l'envoi de message, cette valeur est en effet égale au nombre de destinataires de l'envoi, qui peut être calculé avec le théorème 6.7 si `local_set` contient le port d'origine. Pour pouvoir construire la règle 6.8, nous devons également calculer le nom du sous-programme dans la table des symboles associés au sous-programme généré pour les envois à partir du modèle AADL initial. Celui-ci peut être déduit à partir du nom du thread et de celui du port. Tous les éléments pour le trouver ont déjà été présentés dans le tableau 6.1.

```
theorem get_destination_number

  foreach port in local_set do

    dst := {cnx in connection_set | is_accessing_to (dst, port)};

    requires (unique);

  return (sum (cardinal (dst)));

end get_destination_number;
```

Listing 6.7 – Nombre de threads destinataires

```
subprogram "<thread_name>_<port_name>_interrogatorXn"
   all loops repeats <N> times;
end "<thread_name>_<port_name>_interrogatorXn";
```

Listing 6.8 – Code généré dans le fichier d'assertion

Enfin, une troisième source de code non borné vient de la surcharge engendrée par les opérations d'optimisation - le *Merge*, en particulier. Nous avons vu en 3.4 que dans certains cas de fusion nous générions des modes associés aux différents sous-programmes à exécuter. L'automate contrôlant les changements de mode consiste typiquement en une boucle non bornée dans l'implémentation. Nous savons cependant que le nombre d'itérations le plus pessimiste de cette boucle est égal au nombre de modes présents dans le thread. Nous pouvons donc borner cette boucle avec le listing 6.9, qui fournit un motif d'assertion à ajouter pour chaque thread concerné, où N est le nombre de modes dans le thread.

```
subprogram "<thread_name>_job"
   all loops repeats <N> times;
end "<thread_name>_job";
```

Listing 6.9 – Code généré dans le fichier d'assertion (2)

Construction de la ligne de commande

Nous avons vu que la ligne de commande dépend de trois données variables selon les modèles :

- le nom du programme à analyser ;
- les noms des fichiers d'assertion et TPO ;
- l'architecture cible.

Le premier élément peut être trouvé dans l'architecture, comme illustré dans le tableau 6.1. Le second élément également, puisque le nom du fichier d'assertion est typiquement préfixé par l'architecture concernée, et le nom du fichier TPO par le process que l'on veut évaluer. Finalement, le nom du programme est celui du process associé.

6.2.4 Annotation du modèle AADL

En reprenant le tableau 6.1 à l'envers, nous pouvons retrouver d'abord le process AADL correspondant à l'entrée `program`, puis le thread AADL correspondant à chaque entrée du fichier ESF. Nous savons quel champ nous interesse : nous récupérons donc sa valeur.

Une fois la valeur du WCET du thread obtenu, nous proposons de l'enregistrer dans la property standard AADL *Compute_ Execution_ Time*. Celle-ci contenant un intervalle de durée, nous devons donc trouver la valeur associée à la vitesse du processeur pour traduire la valeur obtenue (en cycles CPU) en durée. Nous présentons dans le listing 6.10 le théorème REAL permettant d'obtenir cette valeur, si `local_set` contient le thread cible.

```
theorem get_processor_speed
  foreach th in local_set do

    process := {p in process_set | (is_subcomponent_of (th, p))};

    cpu := {p in processor_set | (is_bound_to (process, p))};

    requires (unique);

  return (max (get_property_value (cpu, "processor_speed")));
end get_processor_speed;
```

Listing 6.10 – Calcul de la vitesse processeur

Une fois cette valeur calculée, nous pouvons modifier la valeur de la propriété *Compute_ Execution_ Time* du thread cible (ou créer cette dernière, si elle n'existe pas). La borne inférieure de la propriété est laissée inchangée si elle existe déjà, et égale à la borne supérieure dans le cas contraire. La borne supérieure reprend la valeur calculée.

6.2.5 Autres outils

Nous avons donc conçu et implémenté un processus d'enrichissement du modèle autour de l'outil Bound-T, en utilisant les informations extraites du modèle pour la configuration de l'outil et l'interprétation des résultats. Ce processus peut être généralisé, d'abord pour interpréter d'autres informations des analyses de Bound-T, ensuite pour utiliser d'autres outils dans notre processus d'enrichissement. Des outils d'analyse de l'ordonnançabilité, de prédiction sur la taille des files d'attentes ou d'analyse de l'utilisation des bus, en particulier, trouveraient un intérêt particulier dans le cadre de notre processus d'optimisation.

6.3 Vérifier la cohérence des modèles avec REAL

6.3.1 Les contraintes structurelles

Nous avons vu que REAL pouvait être utilisé pour traduire les contraintes définies dans les spécifications de l'application. Cependant, il ne s'agit pas du seul type de contraintes que doivent vérifier les versions optimisées du modèle. En effet, le modèle initial va traduire un certain nombre

de contraintes implicites aux spécifications — par exemple les flots de données, ou encore le typage des diverses connexions. Cette information ne peut être extraite automatiquement des spécifications car celles-ci sont rédigées au moins pour partie en langage naturel — il s'agit alors typiquement du travail de l'architecte.

Cependant, ce travail devrait être limité au minimum, c'est-à-dire être effectué une seule fois, pour limiter non seulement les coûts de développement, mais surtout les risques d'incohérences inhérentes au travail humain, surtout répétitif. Or, dans une approche par optimisation, il est nécessaire d'offrir un mécanisme pour assurer que les contraintes structurelles sont conservées entre les différentes versions du modèle. Vérifier cette cohérence demande typiquement deux fois le même travail : d'une part la traduction explicite des contraintes structurelles de l'application — obligatoirement à partir du modèle initial, pour les raisons sus-citées — et d'autre par un mécanisme permettant de les vérifier dans les modèles optimisés.

6.3.2 Génération de théorèmes

Des théorèmes REAL dans les modèles optimisés peuvent assurer que ceux-ci sont conformes aux contraintes structurelles, une fois ces contraintes connues. D'autres théorèmes — en grande partie factorisables avec les précédents — peuvent extraire ces contraintes du modèle initial. Nous proposons donc une approche permettant de déduire automatiquement les théorèmes REAL vérifiant les contraintes structurelles issues du modèle initial sur les modèles optimisés, en générant celles-ci.

Nous définissons donc une approche en deux étapes, applicable à toute contrainte structurelle :
- Calcul de l'invariant ;
- Génération du théorème.

Pour factoriser le code REAL, nous supposons que nous possédons un théorème REAL *compute_constraint_value*, calculant la valeur d'une contrainte structurelle. Nous ne faisons pas d'hypothèse sur la portée d'une contrainte structurelle : celle-ci peut s'exercer aussi bien au niveau du système qu'au niveau des process ou threads — c'est pourquoi le paramètre unique de toutes les fonctions de calcul d'invariant doit être `local_set`.

Calcul de l'invariant

Le théorème 6.11 donne la fonction finale de calcul d'invariant qui sera utilisée pour chaque composant impliqué du modèle. Ce théorème, appelé de façon externe sur le modèle initial (par exemple par un script shell, voir la section 6.1 pour savoir comment appeler un théorème REAL avec l'API de Ocarina), retourne la valeur de l'invariant dans ce modèle. Cette information est enregistrée dans une variable (nous considérerons par la suite qu'il s'agit de `CONSTRAINT_INVARIANT`).

```
theorem compute_invariant
  foreach set in local_set do

    var inv := compute compute_constraint_value (set);

  return (inv);

end compute_invariant;
```

Listing 6.11 – Calcul de l'invariant

Génération de la contrainte structurelle

Générer le théorème exprimant la contrainte structurelle dans les modèles optimisés peut ensuite être fait sans calcul supplémentaire, et avec un minimum de code REAL généré. Comme le théorème *compute_constraint_value* permet d'obtenir la valeur de l'invariant pour le modèle courant (optimisé), et que nous connaissons la valeur attendue pour celui-ci, il suffit de générer le code REAL illustré par le théorème 6.12, et l'ajouter en annexe des composants AADL concernés. Nous nommons `build_test_theorem` l'application qui pour une contrainte définie par *compute_constraint_value* et sa valeur construit un tel théorème.

```
theorem test_invariant
  foreach set in local_set do

    var actual_inv := compute compute_constraint_value (set);

  check (actual_inv = CONSTRAINT_INVARIANT);

end test_invariant;
```

Listing 6.12 – Vérification de l'invariant

Ajout de la contrainte en annexe

Nous avons évoqué le fait que les théorèmes puissent s'appliquer à plusieurs niveaux du modèle. Il faut donc avoir un mécanisme qui permette d'associer les théorèmes applicables pour chaque type de composants. Nous utilisons l'algorithme 3, qui va sélectionner les contraintes structurelles en fonction de leur préfixe, et les appliquer à tous les composants du modèle dont le type correspond au préfixe en question. A partir de ces éléments, nous proposons l'approche suivante :

- toutes les contraintes structurelles sont ajoutées à une bibliothèque REAL ;
- chaque contrainte structurelle est préfixée par le type de composant à laquelle elle s'applique ;
- on applique l'algorithme 3 sur le modèle initial ;
- les phases d'optimisation ultérieures font hériter les composants modifiés (threads, process...) des contraintes structurelles de leurs parents.

On notera que nous faisons l'hypothèse que toutes les contraintes structurelles sont symétriques — c'est-à-dire qu'elles s'appliquent à tous les composants d'un même type, et que la valeur de l'invariant ne change pas. Il est possible de décrire plus finement les contraintes structurelles en levant cette hypothèse, mais il faut alors trouver des règles d'association pour les contraintes dans les composants transformés (par exemple quand deux threads sont fusionnés). Nous n'avons pour l'heure pas de solution uniforme pour régler cette situation.

6.3.3 Un exemple de contrainte structurelle

Les connexions entre les différents threads dans le modèle initial représentent les flots de données de l'application. Dans le modèle optimisé, cette donnée devrait être invariante quelle que soit la suite d'opérations effectuée, sans quoi le modèle aura été altéré.

Input: System S
Input: Theorem_List TL
Output: System $S2$
$S2 \leftarrow S$ **forall** $t \in TL$ **do**
  if $Is_Prefix(system_, t)$ **then**
    $CONSTRAINT_INVARIANT \leftarrow Call(t, S)$
    $t2 \leftarrow build_test_theorem(t, CONSTRAINT_INVARIANT)$ **forall** $C_{system} \in Systems(S2)$ **do**
      $Annexes(C_{system}) \leftarrow Annexes(C_{system}) \cup t2$
    end
  end
  if $Is_Prefix(process_, t)$ **then**
    $CONSTRAINT_INVARIANT \leftarrow Call(t, S)$
    $t2 \leftarrow build_test_theorem(t, CONSTRAINT_INVARIANT)$ **forall** $C_{process} \in Processes(S2)$ **do**
      $Annexes(C_{process}) \leftarrow Annexes(C_{thread}) \cup t2$
    end
  end
  if $Is_Prefix(thread_, t)$ **then**
    $CONSTRAINT_INVARIANT \leftarrow Call(t, S)$
    $t2 \leftarrow build_test_theorem(t, CONSTRAINT_INVARIANT)$ **forall** $C_{thread} \in Threads(S2)$ **do**
      $Annexes(C_{thread}) \leftarrow Annexes(C_{thread}) \cup t2$
    end
  end
end

Algorithm 3: Ajout des annexes de contraintes structurelles

L'opération merge ne devrait ainsi avoir aucun impact sur le nombre de connexions, puisqu'une connexion sortante ou entrante est soit transformée en auto-connexion, soit conservée. L'opération Move peut avoir un impact, puisqu'elle peut éclater une connexions locale à un process en trois connexions, dont deux sont locales à des process et l'autre locale au system. Pour prendre en compte ce biais, nous proposons comme invariant l'opération suivante : *nombre de connexions locales au process − nombre de connexions locales au system.*

```
theorem compute_inter_thread_connections_number
  foreach s in system_set do

      proc_set := {p in process_set | (is_subcomponent_of (p, s))};

      thread_set := {t in thread_set | (is_subcomponent_of (t, proc_set))};

      inter_thread_cnx_set := {c in connection_set |
                        (is_accessing_to (c, thread_set))};

      inter_process_cnx_set := {c in connection_set |
                        (is_accessing_to (c, process_set))};

  return (msum (cardinal (inter_thread_cnx_set) -
                 cardinal (inter_process_cnx_set)));

end compute_inter_thread_connections_number;
```

Listing 6.13 – Nombre de connexions inter-thread

Le théorème 6.13 calcule cette valeur. On notera qu'on définit les connexions locales à un process comme égale aux connexions accédant à au moins un thread. Cette définition ne prend pas en compte les accès aux objets partagés, et est donc restrictive. Les connexions à des ports en entrée ou sortie sont par contre incluses dans cette valeur.

Nous utilisons ensuite ce théorème pour évaluer le nombre de connexions inter-threads dans le modèle initial. Une fois la valeur récupérée et stoquée dans une variable CNX_NUMBER, nous pouvons générer la contrainte structurelle, qui est ajoutée en annexe du system du modèle optimisé, comme illustré par le théorème 6.14.

```
system top_level

  -- (...)

annex real_specification {**

  theorem test_inter_thread_connections_number

    foreach set in local_set do

      var actual_inv := compute compute_inter_thread_connections_number (set);

    check (actual_inv = CNX_NUMBER);

  end test_inter_thread_connections_number;

  **};
end top_level;
```

Listing 6.14 – Vérification du nombre de connexions inter-thread

6.4 Vue d'ensemble du processus d'optimisation

6.4.1 Un processus d'optimisation à trois voies

La figure 6.8 illustre le processus d'optimisation dans son intégralité. Les ellipses gris clair représentent les différentes étapes du modèle en cours d'optimisation. Les ellipses gris foncé correspondent aux théorèmes REAL. Les rectangles gris médian représentent les étapes faisant elle-même appel à des théorèmes REAL. Finalement, les rectangles gris foncés illustrent les étapes de transformation de modèle au moyen de Ocarina, que ce soit pour générer des modèles (rectangle *transformation de modèles*) ou pour générer du code-source (rectangle *génération de code*).

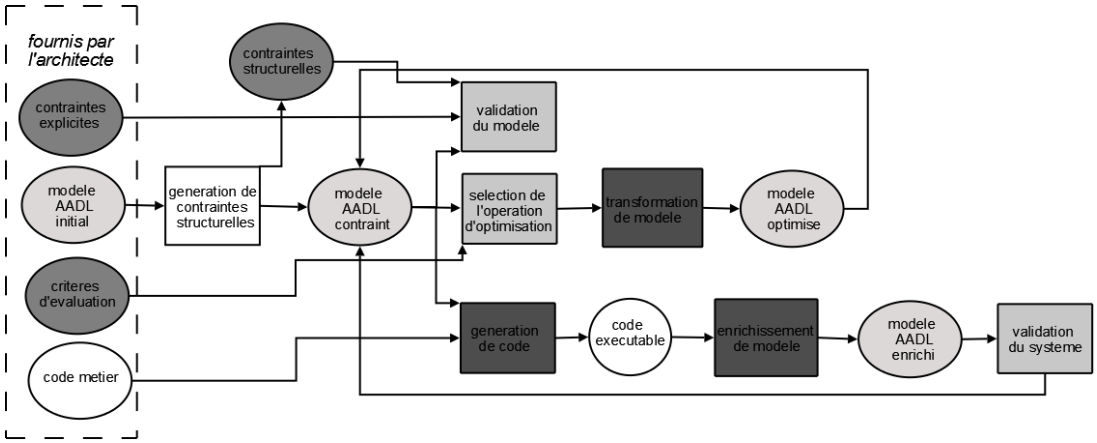


FIG. 6.8 – Implementation du processus d'optimisation à 3 voies

Initialement, l'architecte fournit au minimum le modèle AADL initial et le code-métier associé. Optionnellement, il fournit également des contraintes inhérentes au projet (normalement issues des spécifications) et des critère d'optimisation. Ces deux entrées se font sous la forme de théorèmes REAL. Par rapport au schéma présentant le processus d'optimisation dans le chapitre 1 (voir en 1.3), on remarquera que l'étape de génération des contraintes a été ajoutée avant le début de l'optimisation. Après cette étape, le processus d'optimisation se divise en trois voies :

- optimisation du modèle ;
- enrichissement du modèle ;
- validation du modèle.

Dans les parties suivantes, nous décrivons chacune de ces voies.

6.4.2 Optimisation du modèle

Par optimisation du modèle, nous entendons l'application d'une opération de transformation sur celui-ci qui permette d'améliorer l'évaluation de celui-ci. Pour évaluer l'intérêt d'une opération, le module Ocarina `Transfo.eval` utilise les théorèmes REAL fournis par l'architecte ou les théorèmes utilisés par défaut. Les détails de cette évaluation ainsi que la façon dont l'ensemble des solutions possibles est parcouru sont exposés dans le chapitre 4.

Une fois l'opération sélectionnée, celle-ci est appliquée au modèle, et un modèle dit *optimisé* est produit. Les effets de chaque opération sont discutés dans le chapitre 3. Ce modèle optimisé remplace alors le modèle courant (*modèle contraint* dans le schéma), d'où il peut ré-effectuer une passe d'optimisation ou exécuter l'une des autres voies, selon l'algorithme d'optimisation utilisé.

6.4.3 Enrichissement du modèle

L'étape d'enrichissement du modèle permet de mettre à jour certaines propriétés non-fonctionnelles d'un modèle optimisé susceptibles de diverger de leurs valeurs calculées au cours de l'optimisation. Cette étape, qui utilise le code-métier fournit par l'architecte, doit d'abord générer le système associé au modèle courant. Pour ce faire, il utilise la dorsale appropriée de l'outil Ocarina. Le code-source produit va être ensuite compilé (étape non présentée dans le schéma, incluse dans *génération de code*), et analysé à l'aide d'outils externes (Bound-T, size..). Ces outils sont configurés à partir des information présentes dans le modèle AADL, puis les informations extraites sont réinjectées dans le modèle AADL pour créer un nouveau modèle, dit *modèle enrichi*. La section 6.2 présente ce processus. Le modèle enrichi remplace ensuite le modèle AADL courant.

6.4.4 Validation du modèle

L'étape de validation de modèle correspond en fait à deux étapes distinctes : d'une part la validation des contraintes structurelles (déduites du modèle) ou explicites (définies par l'architecte), qui se répète régulièrement, potentiellement après chaque phase d'optimisation. Cette étape ne fait appel qu'à l'interpréteur REAL, et vérifie que le modèle courant est conforme aux théorèmes représentant ces deux types de contraintes. La façon dont un modèle REAL est décrit et interprété est présentée dans le chapitre 5.

D'autre part, l'étape de validation peut s'appliquer en conjonction avec une étape d'enrichissement du modèle courant. Dans ce cas, elle peut non seulement reprendre les éléments de la version précédente, mais également procéder à des évaluations plus précises — qui relèvent de l'EBB selon la terminologie établie dans le chapitre 4. Dans le cas particulier de l'analyse de l'ordonnancement, c'est dans cette étape qu'elle a lieu. Pour cela, nous utilisons l'outil Cheddar, qui exploite les informations extraites du modèle AADL enrichi.

6.5 Conclusion et avancement

Nous avons présenté un processus permettant de couvrir l'optimisation et la validation des modèles AADL, en apportant une attention particulière à la flexibilité offerte à l'architecte et la cohérence entre les différentes étapes de spécification et vérification. Pour chacune des étapes qui le compose, nous proposons une implémentation.

Dans notre plate-forme basée sur l'outil Ocarina, nous avons implanté un parseur/interpréteur pour le langage REAL. A partir de ce langage, nous proposons une solution pour la vérification et l'évaluation des modèles AADL. Nous avons également implanté un exemple de processus d'enrichissement utilisant les modèles AADL et l'outil d'analyse de WCET Bound-T. Finalement, nous avons implanté deux opérations d'optimisation, ainsi que les algorithmes qui permettent de diriger leur application.

Au moment de la rédaction de ce mémoire, l'étape de validation comptait encore certaines parties manuelles (en particulier l'appel à Cheddar). Le processus d'enrichissement, s'il est implanté, est également lancé manuellement.

Dans le chapitre suivant, nous présenterons un cas d'étude où un modèle initial non-trivial est optimisé en utilisant les diverses étapes de notre processus.

7

Cas d'étude et expérimentation

7.1 Generic Avionic Platform

L'exécutif du langage Ada a été conçu pour les systèmes temps-réel critiques, et supporte des fonctionnalités telles que la gestion de la concurrence ou les mécanismes de communication inter-threads. Ces fonctionnalités sont disponibles nativement dans le langage, à l'opposé — entre autres — du langage C, où elles sous fournies par une bibliothèque qui accède au système d'exploitation.

Il s'est rapidement avéré, cependant, que ces fonctionnalités étaient jugées insuffisantes par la communauté des systèmes temps-réel [35], en particulier à cause de l'indéterminisme inhérent au dispositif de *rendez-vous*, seul mécanisme de synchronisation alors offert par l'exécutif, mais également à cause du manque d'exemple probant de la faisabilité d'un système temps-réel de taille significative.

Pour pallier à ce dernier point, et pour démontrer la faisabilité d'un tel système, un cas d'étude à été développé à la fin des années 80. Le Software Engineernig Institute à l'université Carnegie Mellon (CMU), le Naval Weapon Center et IBM ont participé à la création de la plateforme générique pour l'avionique (GAP, for *Generic Avionic Plateform*), une plateforme logicielle implémentant les specifications logicielles générales pour l'avionique (*Generic Avionic Software Specification* [68]), dont l'objectif est de spécifier de façon réaliste la partie logicielle du contrôleur de mission (*Mission Control Computer*, MCC) d'un avion d'attaque air/air et air/sol.

Le cas d'étude GAP était initialement implémenté en Ada 83. Des versions ultérieures du langage furent rendues compatibles avec le profile pour la concurrence Ravenscar [44], qui assure que le modèle peut être analysé statiquement. En 2008, l'auteur de [56] propose une modélisation en AADLv1 d'une version mono-processeur de GAP, en vue de démontrer les capacités de vérification que pouvait offrir du code Ada 2005 généré par ce modèle en suivant les prescriptions du standard Ravenscar. Nous avons reécris ce modèle pour passer au cas multi-processeur, et rédigé cette nouvelle version de GAP en AADLv2.

Dans la suite de ce chapitre, nous commençons par présenter la spécification GAP, puis le modèle AADLv2 que nous avons conçu pour celle-ci, en détaillant chaque process ainsi que l'architecture matérielle. Nous définissons et discutons dans une seconde partie les différents paramètres que nous avons utilisés pour mener à bien l'expérimentation, ainsi que les contraintes à vérifier sur les modèles. Finalement, nous présentons et interprétons les résultats.

7.1.1 La spécification GAP

GAP modélise les éléments suivants de la plateforme logicielle :

- la configuration des différentes tâches et de leurs contraintes temporelles ;
- les flots de données et leur typage ;
- la difficulté (en terme de temps de calcul) de la partie fonctionnelle de chaque tâche.

Nous décrivons dans la suite de cette section les systèmes matériels et logiciels tels qu'ils sont définis dans la spécification GAP.

Capteurs et sous-systèmes matériels

Les systèmes avioniques de la génération GAP étaient constitués d'un ensemble de capteurs, de commandes et de sous-systèmes connectés par un bus de données à un processeur maître. Les matériels les plus importants dans le cadre de GAP sont :

Air Data Computer (ADC) : estime l'altitude en fonction de la pression barométrique, ainsi que la vitesse de l'air et des champs magnétiques ;

Inertial Navigation System (INS) : estime la position de l'appareil en utilisant des accélèromètres et des gyromètres, fournit également l'assiette de l'avion ;

Radar : radar actif, permet la détection et la traque de cibles aériennes ou terrestres ;

Radar Altimétrique (RATL) : fournit des informations précises sur l'altitude au sol par échantillonnage magnétique du terrain sous l'appareil ;

Radar Warning Receiver (RWR) : envoie des impulsions électromagnétiques sur la surface de l'appareil de façon à détecter les radars actifs émettant en direction de celui-ci ;

Stores Management Systems (SMS) : l'appareil porte des armes ou des modules (capteurs, réservoires de carburant supplémentaires, etc.) sous les ailes et le fuselage. Ces éléments sont nommés collectivement *stores*. Le SMS permet de les gérer et de les relacher à la demande du pilote, en différée ou non ;

Multi-Purpose Display (MPD) : le MPD est un écran tactile dans le cockpit qui peut être utilisé pour afficher diverses informations choisies par le pilote ;

Hands On Throttle-And-Stick (HOTAS) : le HOTAS désigne l'ensemble des boutons et autres commandes accessibles sur le manche à balai ;

Heads-Up-Display (HUD) : il s'agit d'informations vitales qui sont affichées en transparence sur le casque du pilote, de façon à ne pas obstruer sa vision ;

Keyset inclue tous les boutons susceptibles d'être pressés par le pilote mais n'appartenant pas au HOTAS.

Ces différents éléments sont assemblés selon le motif décrit par le schéma 7.1. Les capteurs (Radar, RATL, RWR, ADC et INS) envoient leurs données au *mission computer* sur le bus série, tandis que celui-ci envoie ses commandes à SMS (ou à des capteurs configurables comme Radar ou RWR) de la même manière. SMS envoie des informations donnant l'état courant des *stores*.

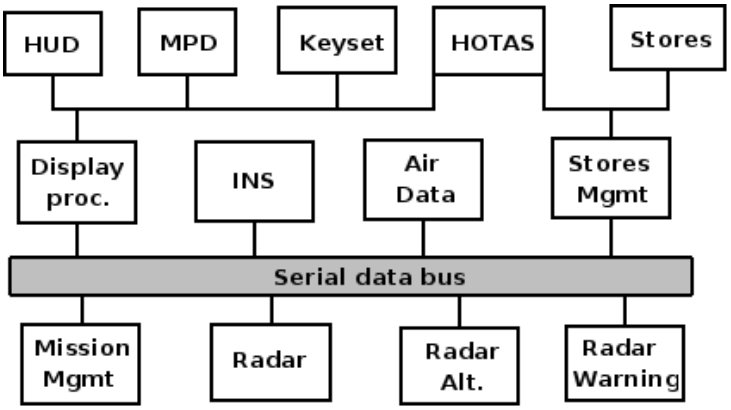


FIG. 7.1 – Architecture du système GAP : décomposition en senseurs et sous-systèmes, de [69]

Sous-systèmes logiciels

Nous décrivons dans cette partie les différentes tâches à effectuer par le MCC. Du point de vue du schéma 7.1, le MCC recouvre à la fois *Display proc.* (pour l'affichage des différents valeurs des capteurs et états des sous-systèmes) et *Mission Mgmt.* (pour la gestion des commandes effectivement déclenchées par le pilote ou le système). Bien que la spécification GAP elle-même ne propose pas de découpage de ces tâches en systèmes, nous reprenons ici celui proposé par les mêmes auteurs en [69], qui propose les principaux sous-systèmes suivants :

Display Met à jour le MPD et le HUD. Reçoit de nombreuses informations des autres tâches, en particulier l'assiette de l'appareil, le statut des *stores*, des différents équipements matériels, etc. L'affichage de ces informations est effectué sur le HUD ou sur le MPD selon le mode ;

Radar Warning Receiver (RWR) Reçoit les données de RWR et contrôle la configuration de ce dernier en fonction des informations saisies par le pilote ;

Radar control and tracking Utilise les informations du radar pour calculer la position et l'orientation d'un éventuel appareil cible. Peut également configurer le radar pour qu'il suive une cible particulière ;

Navigation (NAV) Calcule la position et l'orientation de l'appareil, si le pilotage automatique est activé, calcule également le cap ;

Weapon Sur activation, met à jour le calcul des trajectoires par rapport aux cibles désignées par le sous-système Radar. Permet le lancement des missiles sur commande ;

Built-in-test (BIT) Effectue des diagnostiques sur les différents systèmes.

Le tableau 7.1 donne le détail des contraintes temporelles pour chaque tâche des sous-systèmes décrits ci-dessus. Les tâches dont la période est préfixée par "A" (en l'occurence, toutes celles du sous-système *weapon*) sont sporadiques. GAP couvre donc bien le cas d'un système temps-réel asynchrone. Il est également important de noter que si les valeurs décrites pour WCRT correspondent essentiellement aux temps de réponse des capteurs et autres sous-systèmes matériels (et

Subsystem	Task	Periode/Réponse (ms.)	WCRT	Utilisation
Display	Status Update	200	3	1.50
	Keyset	200	1	0.5
	Hook Update	80	2	2.50
	Affichage	80	9	11.25
	Stores Update	200	1	0.50
RWR	Contact Mgmt	25	5	20.00
Radar	Target Update	50	5	10.00
	Target Filter	25	2	8.00
NAV	Nav. Update	59	8	13.56
	Steering	200	3	1.50
Tracking	Target Update	100	5	5.00
Weapon	Weapon Select	A 200	1	0.50
	Weapon Release	A 200	3	1.50
	Weapon Trajectory	A 50	3	6.00
BIT	Status update	1000	1	0.10
Data Bus	Data polling	40	1	2.50

TAB. 7.1 – GAP : contraintes temporelles

donc sont relativement indépendants de l'implémentation), la valeur *Utilisation* correspond elle à une implémentation monolithique de GAP, et ne peut donc pas être conservée dans le cadre d'une implémentation multi-processeurs.

7.1.2 Modélisation avec AADL

Pour présenter le modèle GAP, nous utiliserons par la suite une version simplifiée de la représentation standard graphique de AADL. Avant cela, nous présentons quelques éléments de notre modélisation des différents composants en fonction de notre problématique.

Les systems

Le `system` est un composant primordial dans la description de l'application, car il sert a spécifier son déploiement. Un mécanisme est donc proposé pour associer les composants matériels aux composants logiciels, et les composants logiciels entre eux.

```
system implementation RAP.GENERIC_IMPL
subcomponents
  soft_main : process Monolithic_RAP.impl;
  soft_weapons : process Weapons.impl;
  soft_navi : process Navigation.impl;
  hardware : process Hardware_Emulator.Impl;
  s_cpu : processor cpu.impl;
  h_cpu : processor cpu.impl;
  mem_1 : memory ram;
  mem_2 : memory ram;
  mem_3 : memory ram;
  the_bus : bus C_Bus.Impl;

connections
  bus access the_bus -> s_cpu.B;
  bus access the_bus -> h_cpu.B;
```

```
  -- Weapons to displays

  c0 : port soft_weapons.Delivery_Mode_Selected ->
       soft_main.Delivery_Mode_Selected
         {Actual_Connection_Binding => (reference (the_bus));};
  -- (...)
properties
  Actual_Processor_Binding => (reference (s_cpu)) applies to soft_weapons;
  Actual_Processor_Binding => (reference (s_cpu)) applies to soft_navi;
  Actual_Processor_Binding => (reference (s_cpu)) applies to soft_main;
  Actual_Processor_Binding => (reference (h_cpu)) applies to hardware;

  Actual_Memory_Binding => (reference (mem_1)) applies to soft_navi;
  Actual_Memory_Binding => (reference (mem_2)) applies to soft_main;
  Actual_Memory_Binding => (reference (mem_3)) applies to soft_weapons;

end RAP.GENERIC_IMPL;
```

Listing 7.1 – Un system AADL : implémentation (extrait)

On peut voir dans le modèle 7.1 que les composants logiciels et matériels sont décrits dans la partie `subcomponents` du system. Le seconds sont ensuite liés au premiers dans la partie `properties`, grâce aux propriétés de type `Actual_Processor_Binding` ou `Actual_Memory_Binding`, sauf dans le cas des bus qui sont eux associés explicitement à chaque connexion logicielle dans la partie `connections`.

Les processes

Les `process` AADL représentent un espace d'adressage. En tant que tels, ils doivent être associés à un processeur et à une mémoire. Ils peuvent potentiellement partager ces ressources. Dans notre modélisation, nous n'assignons pas de propriété particulière aux `process`, si ce n'est la propriété *No_Optimization*, servant à spécifier qu'on ne cherche pas à optimiser un process. Le listing 7.2 illustre un exemple de process dans notre modèle.

```
process Hardware_Emulator
features
  Radar_Mode               : in out data port RAP_Int_32;
  Contact_Number_to_Track  : in out data port RAP_Int_32;
  Contact_Table            : out data port Contact_Table_T.Impl;
  Target_Position          : out data port Location_T.Impl;
  Status                   : in data port RAP_Int_32;

  -- (...)
properties
  Transformations::No_Optimization => true;
end Hardware_Emulator;
```

Listing 7.2 – Un process AADL : déclaration (extrait)

Les threads

```
thread Weapon_Release_T
features
  Time_to_go_to_Release    : in data port RAP_Int_32;
  Weapon_Release_Interval  : in data port RAP_Int_32;
  Delivery_Mode_Selected   : in data port RAP_Int_32;
  Bomb_Button_Depressed    : in data port RAP_Int_32;
```

```
  Manual_Weapon_Release : in event port;
  Release_Signal : out event port;
  Relaunch     : in event port;
  Do_Relaunch : out event port;
properties
  Dispatch_Protocol       => Sporadic;
  Period                  => 200 ms;
  Deadline                => Period;
  Compute_Execution_Time => 0 ms .. 3 ms;
  Cheddar_Properties::Fixed_Priority => 7;
end Weapon_Release_T;
```

Listing 7.3 – Un thread AADL : déclaration

Le thread 7.3 est un thread représentant une tâche sporadique de période 200 ms. On peut voir que les caractéristiques non-fonctionnelles (ici, l'aspect temporel de l'application) sont spécifiées dans le champ `properties`. Ces propriétés peuvent avoir une influence sur le reste du modèle : en tant que thread sporadique, le thread est obligatoirement muni d'au moins un port d'évènements entrant. Dans l'exemple donné, il en possède deux : `Manual_Weapon_Release` et `Relaunch`. Le code métier est associé à ces entrées dans l'implémentation du thread, qui associe une call sequence à chacun des *in event ports*. Le listing 7.4 illustre la façon dont cela est fait à travers la propriété *compute_entrypoint_call_sequence*.

```
thread implementation Weapon_Release_T.i
calls
   cs1 : {
     job1 : subprogram WR_On_Manual_Weapon_Release;
   };
   cs2 : {
     job2 : subprogram WR_On_Relaunch;
   };

-- (...)

properties
   compute_entrypoint_call_sequence   => reference (cs1)
      applies to Manual_Weapon_Release;
   compute_entrypoint_call_sequence   => reference (cs2)
      applies to Relaunch;
end Weapon_Release_T.i;
```

Listing 7.4 – Un thread AADL : extrait de l'implémentation

7.1.3 Le modèle architectural de GAP

Nous présentons ici les différents choix qui ont été fait lors de la réalisation d'une version multi-processeurs de GAP, modélisée en AADLv2. Notre architecture présente un certain nombre de modifications par rapport à la spécification et à sa version proposée en [56] :

- remplacement de l'architecture matérielle par un process qui l'émule ;
- limitation du nombre de connexions ;
- utilisation de la norme AADLv2 comme langage de modélisation [8].

Nous avons choisi de découper la partie logicielle du modèle en 3 process — nous n'avons donc pas repris le découpage présenté dans la section précédente. Ce découpage initial suit une logique fonctionnelle assez triviale, définie par les `process` AADL suivants :

[8] Celui-ci faisant 2279 lignes, nous ne le présenterons pas dans son intégralité.

Displays Regroupe les threads responsables de l'affichage, mais également de la saisie des commandes, ainsi que les threads gérant le radar et le RWR ;

Navigation Regroupe le module de calcul de position et le module de pilotage automatique ;

Weapons Regroupe les threads commandant au fonctionnement des armements : sélection des armes, calcul des trajectoire et mise à feu.

Nous présentons séparément chacun de ces process, avec une vue simplifiée des connexions (une seule connexion de chaque catégorie entre deux threads ou process).

Le process Displays

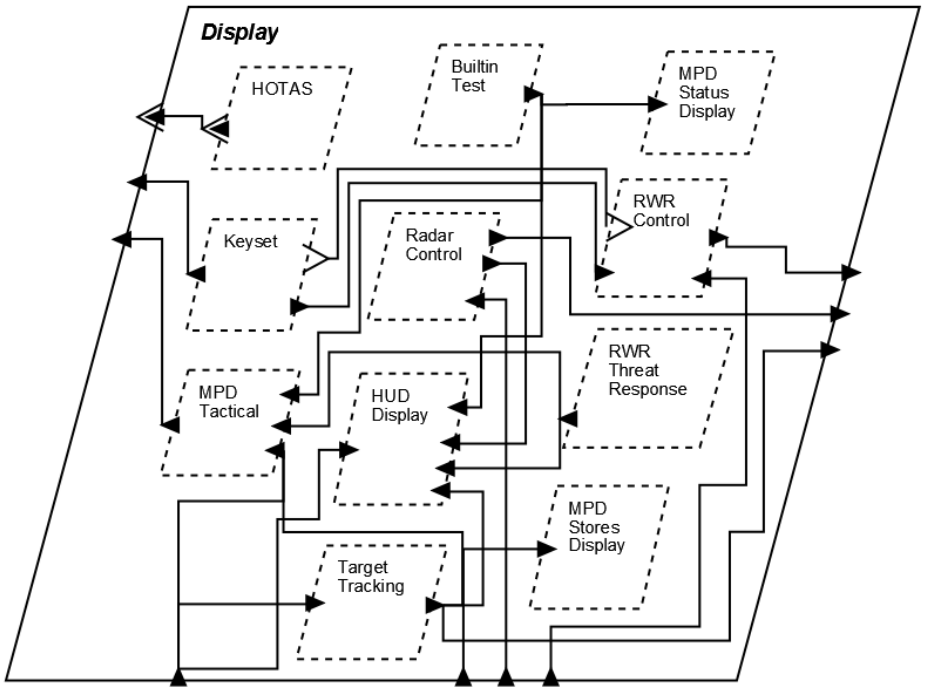


FIG. 7.2 – Le process Display

Le process display, illustré par le schéma 7.2, contient 11 threads, dont 10 périodiques et un sporadique (*RWR_Control*, qui est déclenché sur réception d'un évènement en provenance de *Keyset*).

Le process Weapons

Le process weapons, illustré par le schéma 7.3, contient 3 threads sporadiques. On peut remarquer que deux d'entre eux, *weapon_ trajectory* et *weapon_ release*, ont des ports `in event` connectés à des ports `out event` du même thread. Cette modélisation permet de simuler le fait que le comportement du thread est généralement sporadique, mais devient périodique après son activation, jusqu'à ce que sous l'effet d'un changement de mode il redevienne sporadique.

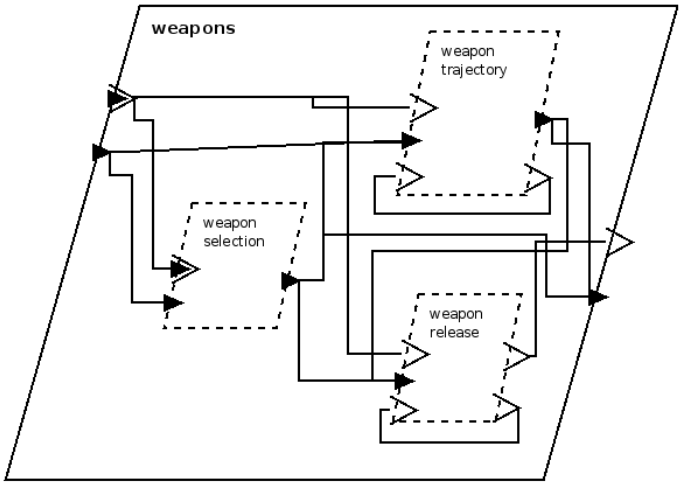


FIG. 7.3 – Le process Weapons

Dans cette implémentation, la réception d'un évènement sur un port va réveiller le thread, qui va ensuite s'envoyer un évènement (sur le second port), lequel évènement réveillera le thread après une durée égale à son intervalle minimal (MIAT, pour Minimum Inter-Arrival Time).

Le process Navigation

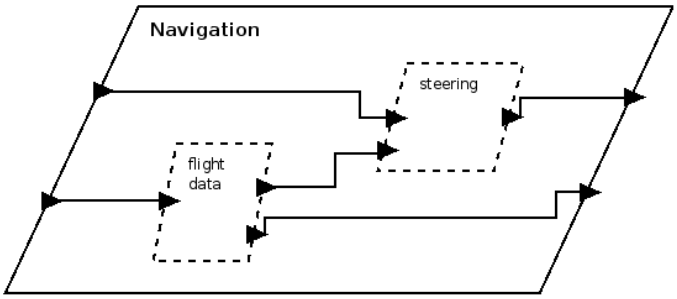


FIG. 7.4 – Le process Navigation

Le process navigation, illustré par le schéma 7.4, ne contient que 2 threads, tous périodiques.

7.2 Procédure de tests

Nous avons vu en 4.2 que nous avions deux algorithmes pour diriger l'optimisation. Nous nous proposons d'évaluer la faisabilité et l'efficacité de ces algorithmes selon leur configuration, en faisant varier cette dernière.

Nous définissons donc dans cette section la configuration des différents paramètres de la procédure d'optimisation, ainsi que les critères que nous avons utilisés pour évaluer son résultat.

7.2.1 Architecture cible

Le code source généré a été compilé pour des processeurs ERC32, une version résistante aux radiations du processeur SPARCv7. Ce processeur est utilisé dans le domaine de l'industrie spatiale. Il est supporté par un certains nombre de compilateurs et de systèmes [18]. Nous générons un exécutif auto-suffisant (incluant le système d'exploitation), grâce au système d'exploitation *Open Ravenscar Real-time Kernel* [93] (ORK). Ce dernier utilise une version adaptée du cross-compilateur GNAT disponible sur plate-forme i386/Linux vers ERC32.

7.2.2 Paramétrage des algorithmes

Les deux algorithmes d'optimisation sont tous deux dotés d'une fonction de coût et d'une fonction de valeur pour les opérations (c'est-à-dire pour l'EBO, selon la terminologie développée en 4.4). Dans le cas de l'algorithme FGH (décrit dans le chapitre 1), nous avons également besoin d'une relation d'ordre entre les threads, pour pouvoir effectuer un tri de ceux-ci. Comme nous l'avons montré, nous pouvons configurer ces fonctions grâce à des théorèmes REAL. Pour démontrer cette possibilité, nous avons effectué deux processus d'optimisation distincts en utilisant des fonctions de valeur différentes : une fonction dite *basée sur les périodes* (PB) et un fonction dite *basée sur les connexions* (CB). Le choix de ces critères est essentiellement motivé par la volonté de démontrer la modularité de notre approche : en aucun cas nous ne prétendons qu'ils soient optimaux.

Dans notre cas d'étude, nous n'avons pas pris en compte la gestion des priorités : chaque thread garde une même priorité durant son exécution, quelle que soit la call sequence qu'il est en train d'exécuter. En effet, l'architecture actuelle du code généré rendait problématique la génération du code associé. Assurer que la solution initial est ordonnançable dans un système non préemptif permet donc d'assurer l'ordonnançabilité du système optimisé, puisque l'impact de la fusion sur l'ordonnancabilité du système va être de réduire le nombre de préemptions possibles. Il s'agit d'une politique conservatrice, puisque le système optimisé n'éliminera qu'une partie des préemptions possibles dans le système original.

Dans cette partie, nous justifions et décrivons ces deux critères, théorèmes REAL à l'appui. Nous justifions et fournissons également la relation d'ordre entre les threads nécessaire à l'algorithme FGH.

Fonction de valeur basée sur les connexions

Nous avons vu que le principal facteur susceptible de modifier l'évaluation du WCET est la suppression des connexions inter-threads. Dans une moindre mesure, le coût de l'ordonnancement local (en terme d'ajouts de *context_ switchs*) a également un certain impact.

```
--  Calcule la reduction de WCET due a la suppressions de
--  communications inter-threads

theorem inter_thread_connections_variation

  foreach thr in local_set do

     emiter_cnx := {cnx in connection_set | is_accessing_to (cnx, thr)};

     others := {th in local_set | is_accessed_by (th, emiter_cnx)
                                and th <> thr};

     inter_cnx := {cnx in emiter_cnx | is_accessing_to (cnx, others)
```

```
                                    and is_accessing_to (cnx, thr)};

     var connection_cost := 5500;
     -- Comes from empirical studies

     var total_transmission_cost :=  connection_cost *
                                     Cardinal (inter_cnx);

  return (Msum (total_transmission_cost));

end inter_thread_connections_variation;
```

Listing 7.5 – Fonction de valeur : coût des connexions supprimées

```
-- Calcule une borne superieure pour le cout de l'ordonnancement
-- local

theorem intrathread_scheduling_cost_variation

   foreach s in system_set do

     tmp := {l in Local_Set | (1 = 1)};

     var thread_min := min (property (tmp, "Period"));

     var thread_gcd := gcd (property (tmp, "Period"));

     var relative_cost := ((thread_min / thread_gcd) -1) *
                          (1000.0 / thread_min);

     var scheduling_cost := 200.0;
     -- Actual cost of a context switch on ERC32/Ork systems

     var balancing_factor := 1.0;
     -- We want to exagerate the overhead due to non-equals deadlines

   return (MSum (relative_cost *
                 scheduling_cost *
                 balancing_factor));

end intrathread_scheduling_cost_variation;
```

Listing 7.6 – Fonction de valeur : coût des context switch ajoutés

La première valeur peut être estimée grâce au théorème REAL du listing 7.5. On peut obtenir une valeur approchée de la seconde valeur en calculant la différence entre le minimum des périodes des threads d'origine et le *quantum* du thread fusionné (donc le PGCD des périodes des threads d'origine). Le listing 7.6 donne le code REAL correspondant à ce calcul. Finalement, nous assemblons ces deux indicateurs avec le théorème présenté dans le listing 7.7.

```
-- Impact estime du merge des threads contenus dans local_set
-- sur le WCET

theorem compute_wcet_variation

  foreach s in system_set do

    var intra := compute intrathread_scheduling_cost_variation (local_set);

    var inter := compute inter_thread_connections_variation (local_set);

  return (msum (inter - intra));
end compute_wcet_variation;
```

Listing 7.7 – Fonction de valeur basée sur les connexions

Fonction de valeur basée sur les périodes

Nous avons vu que, pour un thread fusionné — et à ordonnanceur constant, le nombre de context switch augmente avec la différence entre la période minimale des threads et leur PGCD. Un autre élément à prendre en compte est que la faisabilité d'un merge diminue au fur et à mesure que le PGCD des périodes des threads candidats au merge diminue. Il en résulte que des merges effectués entre des threads ayant des périodes "proches" (en terme de décomposition en nombre premiers) font moins diminuer la faisabilité du système. Un critère de fusion qui favorise l'exploration de l'espace des solutions est donc de minimiser la distance entre la période des threads fusionnée. Le listing 7.8 est un théorème REAL calculant cette valeur.

```
--  Calcule la distance entre le max des periodes et leur PGCD

theorem compute_merging_cost_variation

  foreach s in system_set do

    set := {s2 in local_set | (1 = 1)};

    var old_threads_max_period := (max (property (set, "period")));

    var new_thread_period := (GCD (property (set, "period")));

    var balancing_factor := 100000;
    --  The balancing factor is put in order to balance the result
    --  with the memory value

    var distance := ((new_thread_period / old_threads_max_period)
                     * balancing_factor);

  return (msum (distance));

end compute_merging_cost_variation;
```

Listing 7.8 – Fonction de valeur basée sur les périodes

Fonction de valeur : influence de l'occupation mémoire

Nous avons proposé des fonctions de valeur prenant en compte le coût en WCET ou la faisabilité future d'une opération. Il nous faut cependant prendre en compte la réduction potentielle de l'empreinte mémoire. A cet effet, nous proposons le théorème REAL 7.9. Le résultat de la fonction d'évaluation sera la combinaison linéaire du résultat de ce théorème et d'un des précédents.

```
--  Compute

theorem memory_consumption_variation

  foreach s in system_set do

    var threads_deleted := compute deleted_thread_number (local_set);

    var thread_variation := if threads_deleted > 0
                            then (threads_deleted - 1)
```

```
                              else 0;

  return (msum (thread_variation * 7));
  -- An AADL thread takes around 70 KB of memory

end memory_consumption_variation;

-- Compute the memory footprint of a given

theorem deleted_thread_number

  foreach t in local_set do

  return (Msum (1));

end deleted_thread_number;
```

Listing 7.9 – Fonction de valeur : calcule de la mémoire

On remarquera que le coût d'un thread en mémoire est considéré comme constant. Il s'agit bien sur d'une simplification : un calcul plus précis devrait prendre en compte les propriétés AADL standard `Source_Data_Size`, `Source_Stack_Size` et `Source_Heap_Size`. En ce qui concerne la seconde valeur, celle-ci peut être calculée à partir d'un outil adapté, ou déduite de la connaissance de la taille de la pile du code métier, et de la surcharge apportée par le processus d'optimisation. La troisième valeur vaut toujours 0, car l'allocation dynamique est interdite par nos options de compilation, en accord avec le profile haute intégrité pour Ada Ravenscar.

Relation d'ordre entre les threads

Finalement, il fallait déterminer une relation d'ordre qui permette d'effectuer un tri sur les threads en fonction d'un thread donné. Ce tri devrait ranger les autres threads du process dans l'ordre de gain potentiel en cas de tri avec le premier thread. Il nous faut donc déterminer une fonction qui permette de déterminer le résultat (en terme de performances) d'une fusion entre deux threads (et non entre N threads, comme dans le cas de la fonction d'évaluation).

Nous proposons une solution similaire à la fonction d'évaluation CB : simplement compter les connexions entre les deux threads. Le listing 7.5 est toujours applicable.

7.2.3 Fonction de coût

En 3.4, nous avons vu qu'un condition de faisabilité pouvait être exprimée sur les opérations Merge. Parmi les critères proposés dans le chapitre 4 pour évaluer cette valeur, nous avons choisi d'évaluer la distance minimum des périodes à la période du nouveau thread (qui dans notre cas sera égale au PGCD des périodes des threads fusionnés). Le théorème REAL que nous utilisons est donné par le listing 7.10.

```
--  Calcule la distance minimum a l'echeance, apres fusion des
--  threads contenus dans local_set

theorem compute_deadline_distance_variation

  foreach s in system_set do

    global original_deadline_distance := compute minimum_deadline_distance;

    set := {s2 in local_set | (1 = 1)};
```

```
    var new_thread_wcet := compute thread_wcet (set);

    var new_thread_deadline := (GCD (property (set, "deadline")));

    var new_distance := new_thread_deadline - new_thread_wcet;

    var res := list (original_deadline_distance,
                     new_distance);

  return (MMin (min (res)));
end compute_deadline_distance_variation;
```

Listing 7.10 – Fonction de coût : Distance à l'échéance

Dans ce théorème, la fonction *thread_wcet* retourne la somme des WCETs des threads présents dans *local_set*, tandis que *minimum_deadline_distance* calcule la distance minimum à l'échéance du système avant l'opération Merge.

7.2.4 Fonction d'évaluation de Move

Les interactions du Move avec les performances n'ont pas été étudiées en détail dans le cadre de notre thèse, et les difficultées liées au placement des threads dans les processes ne peuvent pas être résolues par une exploration complète des solutions, s'agissant d'un problème NP-complet. Comme montré dans le chapitre 2, il s'agit d'un domaine sur lequel un certain nombre de recherches ont déjà été menées. Une évaluation plus précise des bénéfices des différents placements possibles devrait s'appuyer sur ces travaux. En particulier, l'approche présentée en [76] constitue une piste prometteuse et s'adaptant bien à notre processus.

Dans notre processus d'optimisation, cependant, nous proposons une approche naive de l'équilibrage de charge : considérant que tous les processus ont la même fréquence, que tous les process disposent du même espace mémoire, et que tous les threads sont identiques en termes d'empreinte mémoire et d'occupation CPU, nous nous contentons d'équilibrer le nombre de threads par process, comme illustré par le listing 7.11. Nous sommes bien sûr conscient du caractère simplificateur de ces hypothèses.

```
--  Calcule la distance en terme de nombre de thread entre les
--  process apres deplacement du thread contenu dans local_set
--  vers le process egalement contenu dans local_set

theorem move_memory_variation

  foreach s in system_set do

    domain := {l in local_set | (1 = 1)};

    thread := {th in thread_set | (is_in (th, domain))};

    src_process := {pr in process_set |
                    is_subcomponent_of (thread, pr)};

    dst_process := {pr in process_set | (is_in (pr, domain))};

    src_proc_threads := {th in thread_set |
                         is_subcomponent_of (th, src_process)};

    dst_proc_threads := {th in thread_set |
                         is_subcomponent_of (th, dst_process)};

    var thread_memory := 10;
    var src_old_memory := cardinal (src_proc_threads) * thread_memory;
    var dst_old_memory := cardinal (dst_proc_threads) * thread_memory;
```

```
    var diff := src_old_memory - dst_old_memory;

  return (msum ((diff)));

end move_memory_variation;
```

Listing 7.11 – Fonction de valeur du Move

7.2.5 Exemple de contrainte sur le modèle

Nous voulons définir un certain nombre de contraintes que devrons respecter toutes les instances du modèle. Dans notre exemple, nous proposons un théorème simple : qu'une mémoire ne soit liée qu'a un unique process. C'est un théorème assurant la non-contamination des espaces mémoires des différents process.

```
theorem one_process_by_memory
  foreach mem in memory_set do

    set := {p in process_set | is_bound_to (p, mem)};

  check (cardinal (set) <= 1);

end one_process_by_memory;
```

Listing 7.12 – Contrainte sur le modèle

Le théorème 7.12 est donc rajouté en annexe du modèle initial et aux modèles optimisés. Il servira à vérifier si la contrainte est bien respectée. On notera que dans ce cas, l'héritage des contraintes entre les différentes versions des modèles se fait automatiquement.

7.2.6 Critères d'évaluation du modèle final

Pour valider notre approche, nous devons définir des critères de performance et de validité pour le modèle optimisé. En premier lieu, les contraintes définies précédement sur les modèles feront bien évidement partie des contraintes de validité finales. A celles-ci on peut ajouter l'ordonnaçabilité.

En ce qui concerne les contraintes de performances, nous n'utiliserons pas les théorèmes développées pour l'EBO puisque celles-ci sont des heuristiques. Pour effectuer une évaluation précise des performances du système, nous devons appliquer une EBB. Cette évaluation devra porter sur les critères sur laquelle s'effectue notre optimisation, c'est-à-dire :

- le WCET global des tâches ;
- l'empreinte mémoire des différents threads.

D'autres critères tels que la consommation énergétique ou le temps de réponse pourraient être pris en compte, et les outils pour une telle analyse peuvent être mis en place. Néanmoins, nous nous limiterons dans le cadre de notre étude à ceux présentés ci-dessus.

Dans le chapitre 6, nous avons défini un processus permettant d'effectuer une analyse du WCET sur un modèle AADL, optimisé ou non. Nous utiliserons cette approche pour estimer l'utilisation du CPU des différents process.

Le calcul de l'occupation mémoire doit prendre en compte la taille de la pile. Nous avons indiqué dans le chapitre 4 comment une telle analyse pouvait être effectuée.

Original	Display	HUD_Display (P, 52ms) + MPD_Tactical (P, 52ms) + MPD_Status (P, 200ms) + MPD_Stores (P, 200ms) + Builtin_Test (P, 1000ms) + Keyset (P, 200ms) + HOTAS (P, 40ms) + Radar_Control (P, 40ms) + Target_Track (P, 40ms) + RWR_Threat (P, 100ms) + RWR_Control (S, 400ms)
	Weapons	Weapon_Selection (S, 200ms) + Weapon_Trajectory (S, 100ms) + Weapon_Release (S, 200ms)
	Navigation	Flight_Data (P, 59 ms) + Steering (P, 80 ms)
Système FGH-CB	Display	(HUD_Display x MPD_Tactical_Display) + (RWR_Control x RWR_Threat_Response)
	Weapons	(Builtin_Test x MPD_Status_Display x Radar_Control x MPD_Stores_Display x Target_Tracking x Keyset x HOTAS) + (Weapon_Release x Weapon_Selection x Weapon_Trajectory)
	Navigation	Flight_Data + Steering
Système FGH-PB	Display	(Radar_Control x Target_Tracking x HOTAS)
	Weapons	(Builtin_Test x MPD_Status_Display x MPD_Stores_Display x Keyset x Weapon_Release x Weapon_Selection x Weapon_Trajectory x RWR_Control x RWR_Threat_Response) + (HUD_Display x MPD_Tactical_Display)
	Navigation	Flight_Data + Steering

Tab. 7.2 – Composition du modèle GAP après optimisation avec FGH

Nous fournissons également des indications sur l'optimisation elle-même, telle que le temps d'exécution du processus et le nombre d'itérations effectuées par l'algorithme. Ces informations donnent une estimation de la complexité effective des algorithmes d'optimisation.

7.3 Résultats et interprétation

7.3.1 Résultats

Nous présentons dans cette partie les résultats des algorithmes d'optimisation, selon les critères de configuration retenus.

Heuristique Gloutonne

Le tableau 7.2 donne le résultat de l'optimisation par l'algorithme FGH, en utilisant successivement comme critère de valeur les connexions (FGH-CB) et les périodes (FGH-PB). Le symbole "x" désigne une composition de threads, c'est-à-dire deux threads fusionnés, tandis que le symbole "+" désigne une addition de threads, c'est-à-dire deux threads concurrents et séparés.

Dans les deux systèmes obtenus respectivement par l'application de FGH-CB et FGH-PB, on observe deux types de différences.

– en premier lieu, un plus grand nombre de thread finaux est obtenu par l'application de la version *basée sur les périodes* de l'algorithme. Ce résultat est cohérent avec la construction de l'algorithme, qui privilégie comme son nom l'indique les périodes, faisant décroître

moins rapidement les périodes des threads fusionnés, ouvrant ainsi la voie à plus de fusions ultérieures, comme expliqué dans le chapitre 4 ;
- ensuite, nous voyons que de larges blocs de threads peuvent être déplacés d'un process à l'autre, dans les deux implantations. Dans ce cas, il s'agit de l'effet de la fonction *Move*.

Outre ces éléments, les deux versions présentent quelques différences, dont nous discutons pour chaque process dans le système fusionné.

Display Avec *FGH*, nous imposons le premier paramètre de l'opération Merge. Le principal critère de décision est le nombre de connexions entrantes ou sortantes. Dans notre modèle, le thread `Builtin_Test` est tout d'abord élu, puisque celui-ci envoie un nombre important de données aux threads `MPD_Status_Display`, `HUD_Display` et `MPD_Stores_Display`. Dans la version *basée sur les connexions*, c'est logiquement avec le premier de ces threads qu'il est d'abord fusionné. `Radar_Control` est également fusionné, car il envoie un message d'erreur à `MPD_Status_Display`, puis `MPD_Stores_Display`, `Target_Tracking` (qui envoie la table des contacts à `MPD_Status_Display`). à ce point de l'algorithme, la période du thread créé est de 40 *ms*. `Keyset` et `HOTAS` sont ensuite fusionnés à ce thread, malgré l'absence de connexions à ce thread, car leur périodes sont également de 40 *ms*. A contrario, `HUD_Display` n'est pas fusionné car sa période de 52 *ms* violerait la condition de faisabilité. Ce dernier est donc fusionné dans un nouveau thread avec `MPD_Tactical_Display`, dont il partage la période. Finalement, `RWR_Control` et `RWR_Threat_Response` fusionnent pour former un troisième thread.

Dans la version *basée sur les périodes*, le critère de connexion est ignoré (sauf pour l'élection du premier paramètre de fusion), au profit d'une mesure de la réduction de la valeur minimale des périodes de l'ensemble de threads candidats à la fusion. Le premier thread créé par un ensemble de Merge contiendra ainsi `RWR_Control` (qui partage avec `Builtin_Test` et `MPD_Status_Display` une basse fréquence de 200 *ms*). Il en va de même, un peu plus tard, pour `RWR_Treat_Response`. `Target_Tracking`, `Radar_Control` et `HOTAS`, tous de période 40 *ms*, qui sont pour leur part fusionnés dans un second thread. Comme dans la version précédente (et pour la même raison), `HUD_Display` et `MPD_Tactical_Display` sont fusionnés dans un troisième thread.

Dans les deux versions de l'algorithme, le premier thread fusionné est déplacé vers le process weapons.

Weapons Les trois threads `Weapon_Selection`, `Weapon_Trajectory` et `Weapon_Release` sont fusionnés car connectés entre eux et de périodes proches (100 *ms* pour les deux premiers, et 200 *ms* pour le dernier). Cette fusion s'opère dans les deux versions.

Dans la version *basée sur les périodes* de l'algorithme, le thread déplacé du process display est fusionné avec le précédent thread, car leur périodes sont identiques (100 *ms*) et que la fusion ne viole pas les règles de faisabilité.

Navigation Les deux threads du process Navigation ont des périodes premières entre elles. La fusion de ces deux threads engendrerait donc un thread de période 1 *ms*, quelle que soit l'implantation de l'algorithme. Comme une telle période violerait évidement la condition de faisabilité, aucune fusion n'est effectuée.

Heuristique Semi-Gloutonne

Le tableau 7.3 donne le résultat de l'optimisation par l'algorithme *HGH*, en utilisant successivement comme critère de valeur les connexions (HGH-CB) et les périodes (HGH-PB).

Original	Display	HUD_Display (P, 52ms) + MPD_Tactical (P, 52ms) + MPD_Status (P, 200ms) + MPD_Stores (P, 200ms) + Builtin_Test (P, 1000ms) + Keyset (P, 200ms) + HOTAS (P, 40ms) + Radar_Control (P, 40ms) + Target_Track (P, 40ms) + RWR_Threat (P, 100ms) + RWR_Control (S, 400ms)
	Weapons	Weapon_Selection (S, 200ms) + Weapon_Trajectory (S, 100ms) + Weapon_Release (S, 200ms)
	Navigation	Flight_Data (P, 59 ms) + Steering (P, 80 ms)
Système HGH-CB	Display	(RWR_Threat_Response) + (HUD_Display x MPD_Tactical_Display) + (RWR_Control x Keyset x Radar_Control x MPD_Status_Display x Builtin_Test x MPD_Stores_Display x HOTAS)
	Weapons	(Target_Tracking) + (Weapon_Release x Weapon_Selection x Weapon_Trajectory)
	Navigation	Flight_Data + Steering
Système HGH-PB	Display	(HUD_Display x MPD_Tactical_Display) + (Builtin_Test x MPD_Status_Display x MPD_Stores_Display x Keyset x RWR_Control x RWR_Threat_Response)
	Weapons	(Radar_Control x Target_Tracking x HOTAS) + (Weapon_Release x Weapon_Selection x Weapon_Trajectory)
	Navigation	Flight_Data + Steering

TAB. 7.3 – Composition du modèle GAP après optimisation avec HGH

L'heuristique semi-gloutonne (HGH) assure une exploration plus large de l'espace des solutions. Paradoxalement, la nature optimale du premier ensemble de threads fusionnés tend à réduire les possibilités de fusions ultérieures. Nous obtenons au final plus de threads — et des threads plus déséquilibrés en terme de charge (WCET/Période), que dans l'algorithme FGH. Nous décrivons ci-dessous les opérations effectuées par l'algorithme HGH, et ce pour les versions *basée sur les connexions* et *basée sur les périodes*.

Display Dans $HGH - PC$, comparativement à $FGH - PC$, le thread initial inclue le thread `RWR_Control`, en plus de tous ceux présents dans le premier thread de $FGH - PC$, à part `Target_Tracking` qui en est exclu. Ce dernier ayant une charge bien plus importante, nous avons donc bien créé un thread plus performant (selon nos critères), et nous vérifions au passage que cette situation correspond au cas où le thread exclu dans $FGH - PC$ est le premier fusionné dans $HGH - PC$. `RWR_Control` et `RWR_Threat_Response` fusionnent ensuite pour former un second thread. Finalement, `RWR_Threat_Response` ne peut plus être fusionné avec aucun thread existant sans violer la contrainte de faisabilité. `Target_Tracking` est ensuite déplacé vers le process Weapons.

Dans la version *basée sur les périodes*, `Target_Tracking`, `Radar_Control` et `HOTAS` sont tout d'abord fusionnés, comme dans $FGH - PC$ et pour les même raisons. De la même façon, `HUD_Display` et `MPD_Tactical_Display` sont fusionnés dans un second thread. Tous les autres threads du process seront fusionnés dans un troisième thread de 100 ms, lequel sera ensuite déplacé vers Weapons.

	FGH-CB	HGH-CB	FGH-PB	HGH-PB
Iterations	536	86	723	101
Durée (s)	4.19	2.63	2.88	3.37
Gain en Mémoire	30%	32%	39%	36%

TAB. 7.4 – Coûts et gains de l'optimization

Weapons Pour la même raison qu'avec l'algorithme précédent, les threads `Weapon_Selection`, `Weapon_Trajectory` et `Weapon_Release` sont fusionnés en un seul thread, et ce quel que soit la version de *HGH*. Aucune fusion ultérieure n'est possible, les threads déplacés ayant déjà une charge trop importante.

Navigation Pour la même raison qu'avec l'algorithme précédent, aucune fusion n'a lieu.

7.3.2 Performances

Nous présentons dans cette partie les différentes mesures que nous avons effectuées sur les modèles générés par nos différents algorithmes, tant du point de vue de leurs performances intrinsèques (mémoire, WCET, ordonnançabilité), que des contraintes définies par l'architecte ou du coût de l'optimisation. En ce qui concerne l'ordonnançabilité, celle-ci à été démontrée avec l'outil Cheddar [83], et est valable pour tous les modèles générés. Les contraintes de l'utilisateur sont automatiquement validées par Ocarina, car elles sont ajoutées en annotations REAL aux modèles optimisés.

Dans notre exemple, nous avons testé la contrainte `one_process_by_memory`, qui vérifie qu'une et une seule mémoire est associée à chaque process. Cette contrainte est testée grâce au théorème 7.13.

```
theorem one_process_by_memory
  foreach mem in memory_set do

    set := {p in process_set | is_bound_to (p, mem)};

  check (cardinal (set) <= 1);

end one_process_by_memory;
```

Listing 7.13 – Théorème one_memory_by_process

Mémoire

Le tableau 7.4 résume le gain en terme d'occupation mémoire des systèmes générés à partir des différents modèles optimisés, en considérant cependant uniquement la mémoire statiquement allouée (voir 7.2.2). Nous voyons que de ce point de vue, l'algorithme $FGH - PB$ est le plus efficace et que, en règle générale, l'approche *basée sur les périodes* permet plus de fusions que l'approche *basée sur les connexions*.

WCET

Nous avons proposé dans le chapitre 4 un processus pour l'évaluation des modèles architecturaux. L'outil utililisé (Bound-T) ne possède pas une granularité assez fine pour exploiter ses

	Original	FGH-CB	HGH-CB	FGH-PB	HGH-PB
Connexions	188	171	155	180	160
N_{It}	66	3	3	6	3

TAB. 7.5 – Optimisation : impact sur le WCET

résultats en l'état. Cependant, nous avons vu dans le chapitre 3 quels sont les éléments influant sur le WCET réel d'une fonction :

- la surcharge de code induite par la construction de l'automate ;
- le nombre de contexte switch ;
- le nombre de connexions inter-threads et inter-process.

Le premier élément se décompose en deux parties : une valeur statique quelle que soit la fusion, correspondant au coût d'une construction `case` dans le langage Ada 2005, d'une boucle controlée par un test, ainsi que d'un appel de fonction déclenchant une simple affectation.

La seconde partie est variable en fonction de la fusion, car il désigne le coût de l'ordonnanceur local de l'automate, appelé à la fin de chaque call sequence. Dans le pire cas, il est appelé N fois lors d'un dispatch, N étant le nombre de threads fusionnés. Le coût de chaque appel est celui d'un appel de fonction, d'une incrémentation d'entier et d'une affectation.

Comme la seconde partie de la surcharge induite par le code, le second élément dépend directement du nombre maximum d'appels de sous-programmes déclenchés lors d'un dispatch. Dans le pire des cas, dans la première version, un thread était préempté une fois par chaque autre thread du même process. Le surcoût variable est donc proportionnel au nombre maximum de thread par process, au carré (puisque chaque thread peut être interrompu). Avec des hypothèses d'ordonnancement RMA, chaque priorité est unique au sein d'un même espace d'ordonnancement (processeur), et un thread ne pouvant être interrompu que par des threads de priorité supérieure à la sienne, on peut borner cette valeur par $N_{It} = \frac{n^2+n}{2}$, avec n comme nombre maximum de threads par process.

Le tableau 7.5 illustre la valeur de N_{It}, ainsi que les variations de connexions pour chaque résultat obtenu. On notera que la fonction de performance de Move ne tient pas compte de ce dernier paramètre, qui est donc surévalué par rapport à une version plus efficace de l'optimisation.

On remarquera que le critère d'optimisation basé sur les connexions donne globalement de moins bons résultats que le critère basé sur les périodes. C'est un résultat intéressant dans la mesure où le dernier a pourtant été créé dans cet objectif. Il est donc globablement plus intéressant de faire le plus de fusions possible, sans se préoccuper des connexions.

Coût de l'optimisation

Le tableau 7.4 indique les coûts respectifs de chaque algorithme d'optimisation utilisé, que ce soit en nombre d'itérations ou en temps d'exécution (sur une architecture double-coeur, cadencée à 2.66 *Ghz*, avec 4 *Go* de mémoire vive).

Les variations entre les valeurs ne sont pas très significatives, car elles sont essentiellement causées par le ralentissement dû à un manque de mémoire, provoqué par la multiplication des instances de modèle en mémoire. Pour pallier ce ralentissement, nous relançons l'algorithme d'optimisation régulièrement de manière à vider la mémoire. Une amélioration future consisterait à réorganiser l'allocation en mémoire des modèles par Ocarina.

On peut cependant noter que toutes les exécutions s'effectuent en quelques secondes, ce qui est un délai plus que raisonnable pour une optimisation statique (*off-line*). Outre prouver l'efficacité de l'interpréteur de REAL, ce résultat sur un système raisonnablement complexe permet

d'envisager l'optimisation de systèmes de plus grande taille, ou la réalisation d'algorithmes d'optimisation explorant une partie plus grande du système (par exemple en testant successivement toutes les fonctions d'optimisation).

7.4 Synthèse

Nous voyons que nous pouvons produire des instances optimisées d'un modèle de système temps-réel embarqué réparti non trivial, et obtenir un résultat plus performant. Nous avons également vu que les critères de ces performances pouvaient être aisément modifiés par l'architecte, grâce au langage REAL. Nous avons également évoqué plusieurs moyens de vérifier a posteriori ou en cours d'optimisation une mesure effective d'un certain nombre de critères de performances.

Les résultats que nous avons obtenu varient en fonction des critères de perfomances. Parmi les critères étudiés, l'heuristique gloutonne basée sur les périodes ($FGH-PB$) est plus efficace pour réduire la mémoire, tandis que l'heuristique gloutonne basée sur les connexions ($FGH-CB$) permet de réduire de façon plus importante le WCET. Les fonctions $HGH-PB$ et $HGH-CB$ obtiennent des résultats moins importants, mais ont un coût moins élevé. Il est également à noter qu'aucune de ces fonctions ne réduit une des valeurs par rapport aux valeurs du modèle initial — le but d'optimisation est donc atteint. Le fait que les profils des résultats obtenus soient différents valide notre thèse initiale : il est nécessaire de pouvoir configurer les fonctions d'optimisation.

8

Vers un processus de développement basé sur les modèles

8.1 Un processus de développement basé sur l'optimisation des modèles

Nous avons vu que l'augmentation en taille et en complexité des systèmes embarqués temps-réel, les contraintes économiques et industrielles toujours plus fortes pesant sur eux et leur diffusion dans un nombre toujours plus grand de secteurs provoque un nombre important de retards de projets et, finalement, leur annulation.

Dans notre mémoire, nous avons présenté un processus permettant de réduire ces retards. En particulier, nous avons proposé un processus qui transforme le développement en une chaine linéaire d'étapes, sans régression possible à une étape antérieure du cycle de développement. Nous avons vu que ce processus ne pouvait être mis en place qu'au niveau du cycle de développement, en se basant sur un langage-pivot, qui permet de décrire l'architecture du système.

En plus du critère principal — garantir l'obtention d'un système fonctionnel au regard des spécifications — nous avons identifié un certain nombre de contraintes que devait respecter ce processus :

- adéquation des critères d'optimisation aux spécifications du système ;
- flexibilité des critères d'optimisation ;
- flexibilité des contraintes sur les modèles ;
- cohérence entre les spécifications et le système réel ;
- validation du système réel.

Nous avons évalué un certain nombre de solutions existantes pour répondre à cette problématique, et sommes arrivé aux conclusions suivantes :

Adéquation de l'optimisation aux spécifications du système Le processus d'optimisation recouvre des actions différentes, parfois contradictoires, selon le contexte dans lequel il est effectué. Ce contexte lui-même peut évoluer durant la durée du projet (par l'émergence de nouvelles technologies ou au contraire l'imposition de nouveaux critères de performances), et les critères peuvent avoir une importance variable selon le fait qu'ils soient ou non remplis à un moment donné du processus d'optimisation.

Il est donc important que la fonction d'optimisation puisse être déduite des spécifications originales et du modèle courant — au moins quant à la pondération de ses différents composants.

Flexibilité des critères d'optimisation Le processus d'optimisation varie en fonction de trois paramètres :

- le modèle en entrée ;
- l'algorithme d'optimisation utilisé ;
- la fonction d'évaluation utilisée.

La plupart des approches existantes ne permettent pas de configurer les deux derniers paramètres. Pourtant, un critère d'évaluation pertinent dans certains contextes peut ne pas l'être pour d'autres (par exemple, la consommation énergétique), indépendamment de l'algorithme d'optimisation utilisé. De même, tous les algorithmes d'optimisation ne sont pas adaptés à tous les problèmes, selon leur taille, la puissance de calcul disponible ou l'exigence en terme de ressources. Un processus d'optimisation devrait donc pouvoir offrir à l'architecte un moyen simple - c'est-à-dire peu coûteux en terme d'apprentissage - de configurer la fonction d'optimisation et la fonction d'évaluation.

Dans le cadre de notre thèse, nous nous sommes concentrés sur le second problème. Nous proposons plusieurs techniques d'optimisation, mais voulons offrir une solution plus flexible pour définir la fonction d'évaluation. Nous avions donc besoin d'un formalisme pour décrire des expressions sur le modèle architectural. Nous avons vu dans notre état de l'art que les solutions actuellement existantes n'était pas satisfaisantes au regard des critères requis.

Flexibilité des contraintes sur les modèles Nous avons vu que dans un certain nombre de cas, des contraintes qualitatives doivent pouvoir être exprimées et vérifiées sur le modèle. Vérifier une propriété dans le code ou dans le comportement du système réel est une opération longue, difficile et sujette à erreur. De plus, l'expression des contraintes est elle-même une opération difficile, qui échappe souvent à la compétence de l'architecte. L'effet est que le travail effectué lors de la modélisation est généralement ignoré lors de l'opération de vérification, générant par là des risques de divergence entre la représentation qu'a le spécialiste des méthodes de vérification du système, et l'intention originelle de l'architecte. Cette divergence peut induire la vérification de propriétés qui sont, en elle-même, erronées.

Pour pallier ce risque, l'étape de modélisation architecturale doit se doubler d'une étape de traduction des contraintes exprimées dans les spécifications. Si le formalisme dans lequel sont exprimées ces contraintes est proche de celui utilisé pour décrire le modèle lui-même, le surcoût d'une telle opération est faible, pour un gain en sûreté important.

Nous avons donc besoin d'un langage d'expression de contrainte basé sur le langage architectural. Comme un tel langage permettrait également d'exprimer des expressions arithmétiques, il pourait être utilisé pour factoriser la contrainte de flexibilité des critères d'optimisation.

Cohérence entre les spécifications et le système réel L'ensemble des contraintes définies précédement doivent s'appuyer sur un formalisme unique qui sera notre langage-pivot, qui permet de représenter notre spécification. Les langages de description architecturale offrent une structure à la fois souple et facilement utilisable pour identifier les différentes composantes d'une application temps-réel embarqué. Parmi ces langages, nous avons choisi AADL dans le cadre de notre thèse.

La génération de code, si elle est basée sur des motifs documentés, permet d'assurer la cohérence entre le modèle architectural et le système réel.

Un langage de description de contraintes sur le modèle architectural permettrait à l'architecte de spécifier des contraintes exprimées dans les spécifications, et de les vérifier dans le modèle architectural. Par conséquent, il permettrait de vérifier la cohérence entre les spécifications et le modèle formel.

Il ressort des deux points précédents que l'association de techniques de génération de code, l'utilisation de modèles architecturaux et l'existence d'un langage de description de contraintes permet d'assurer la cohérence entre les spécifications et le système réel.

On notera que la cohérence n'est assurée qu'au regard des contraintes décrites par l'architecte, c'est-à-dire celles définies initialement dans les spécifications. L'opération d'optimisation, cependant, est susceptible d'avoir un impact sur certains éléments du modèle qui remette en cause certaines hypothèse implicites des spécifications. Pour pallier ce problème, il doit être possible de déduire automatiquement un certain nombre de caractéristiques du modèle initial, et de s'assurer de leur conservation tout au long de l'optimisation. Ces caractéristiques peuvent être analysées grâce à notre langage d'expression de contraintes, et vérifiées à partir d'autres contraintes, générées au début du processus de développement.

Validation du système réel Finalement, un mécanisme doit être proposé pour vérifier que le système réel issu du modèle optimisé répond bien aux exigences initialement décrites en terme de critères d'efficacité. Ce mécanisme doit pouvoir analyser des propriétés locales du système réel (donc de binaires), et les associer dans des calculs pour les comparer aux propriétés attendues. Un exemple de propriété à vérifier peut être l'ordonnançabilité, qui demande d'extraire le WCET de chaque sous-programme, puis d'effectuer un calcul.

L'opération d'analyse des binaires est une opération très spécialisée, qui en soit constitue le sujet d'une ou de plusieurs études. Un tel travail ne peut donc pas être effectué de façon réaliste dans le cadre de nos travaux. Il est donc nécessaire de mettre en place un mécanisme permettant à plateforme d'optimisation de communiquer avec des outils externes. Ces outils nécessitant généralement une configuration, il nous faut offrir un moyen de configurer automatiquement ceux-ci. Nous pouvons extraire toutes les informations nécessaires à cette configuration du modèle architectural. Encore une fois, nous avons besoin d'un langage permettant d'interroger le modèle architectural.

8.2 Contributions

8.2.1 Optimisation

Le sujet principal de notre thèse est l'optimisation des systèmes temps-réel embarqués. Nous présentons ici les contributions que nous avons apporté à ce champ.

Opérations d'optimisation

Après avoir étudié l'impact de différentes transformations de modèle sur les performances du système réel, nous avons défini trois opérations sur les modèles AADL permettant d'explorer l'ensemble des solutions d'un problème (cf. chapitre 2). Avec une restriction sur les modèles en entrée, nous pouvons limiter le nombre de ces opérations à deux. Nous avons fourni une implantation de ses deux opérations dans l'outil Ocarina (chapitre 6).

Algorithmes d'optimisation

Nous proposons deux algorithmes de contrôle pour l'exploration des solutions. Ces deux algorithmes sont basés sur des heuristiques qui estiment la valeur de la solution courante, mais se distinguent par l'espace effectivement visité. Chacun de ses algorithmes se décline en deux versions, selon l'heuristique qu'il utilise pour évaluer la valeur d'une solution (cf. chapitre 4). On

notera que nous avons pris soin de différencier l'implantation des algorithmes de la définition des heuristiques, qui peuvent être configurées par l'architecte.

8.2.2 Langage de description de contraintes

Nous avons vu que la définition d'un langage de description de contraintes spécifique au formalisme de modélisation que nous utilisions (AADL) était nécessaire. Dans le chapitre 5, nous présentons la structure du langage que nous avons conçu dans ce but, REAL, ainsi que des exemples d'application. Dans la partie ci-dessous, nous décrivons les différents usages possibles de ce langage.

Contraintes utilisateur

REAL peut être utilisé pour définir des contraintes spécifiques au projet. Ces contraintes doivent traduire directement les spécifications initiales. Elles peuvent être décrites de façon indépendante du modèle, assurant ainsi la cohérence entre les spécifications et le modèle — il s'agit donc d'une application d'aide à la modélisation. Les contraintes définies dans le chapitre 5 pour vérifier la sécurité dans les modèles AADL sont un exemple d'une telle application. Ces théorèmes sont utilisés dans les projets POK et MOSIC.

Contraintes générées

Des théorèmes REAL peuvent également être générés, de manière à déduire des contraintes de cohérence entre deux modèles. Une telle application est illustrée dans le chapitre 6, et des propositions — non exhaustives — de vérifications sont avancées. Une telle approche permet d'assurer que les différentes opérations d'optimisation conservent les propriétés structurelles du modèle initial.

Critères d'évaluation

Les théorèmes REAL peuvent également être utilisés pour retourner des valeurs non booléennes. Nous utilisons cette possibilité pour décrire les fonctions d'évaluation qui guident l'optimisation avec les théorèmes REAL présentés dans le chapitre 7.

8.2.3 Evaluation

Nous avons vu que l'évaluation était un critère déterminant dans l'efficacité d'un processus d'optimisation, et même dans un processus de développement en général. Nous présentons dans cette partie les contributions que nous apportons concernant cet aspect.

Définition des critères

Nous avons défini dans le chapitre 4 les différents critères qui permettent d'évaluer une architecture. Nous avons montré qu'il était nécessaire dans une approche dirigée par les modèles de définir deux niveaux d'évaluation pour chaque critère identifié (niveau modèle et niveau binaires). Dans le cadre d'une approche munie d'une étape d'optimisation, il faut encore définir un niveau supplémentaire (niveau opération d'optimisation). Nous avons illustré la manière dont ces trois étapes peuvent être insérées au sein d'un processus d'évaluation.

Processus de validation

Nous avons vu dans les étapes précédentes que nous pouvons garantir la cohérence entre les spécifications et le modèle AADL, puis entre les différents modèles architecturaux. Grâce à notre connaissance des motifs de génération, nous pouvons également assurer la cohérence du code généré et du modèle dont il est issu. Notre connaissance de la plateforme intergicielle et de l'architecture matérielle nous permet par ailleurs d'estimer la valeur de caractéristiques non-fonctionnelles (WCET, utilisation mémoire, etc) du système généré lui-même (et non simplement du code qu'il exécute). Cependant, nous ne pouvons pas déduire les valeurs exactes de ces informations par une simple analyse de l'architecture.

Pour pallier ce manque, nous définissons un processus pour valider le système final, en fonction de ses propriétés réelles, telles qu'elles sont mesurées par des outils tiers. Nous utilisons les informations contenues dans le modèle AADL pour configurer automatiquement les outils, ce qui permet d'inclure l'étape d'analyse sur les binaires sans rompre la chaine de développement, et sans intervention de l'architecte.

Ce même processus permet de construire un modèle enrichi, ce qui permet de corriger périodiquement la divergence entre les valeurs estimées des caractéristiques non-fonctionnelles et leurs valeurs réelles dans le système issu du modèle courant. Dans le chapitre 6, nous définissons le principe et l'implantation d'une interface entre le modèle AADL et les outils. Un exemple est donné avec l'analyse du WCET en utilisant Bound-T.

8.3 Perspectives

Nous discutons dans cette section des améliorations qui peuvent être apportées au processus d'optimisation, tant du point de vue de son implantation que des éventuels ajouts qu'on pourrait lui faire.

8.3.1 Améliorations à apporter à l'implantation du processus

En premier lieu, un effort doit être fait pour la définition et l'implantation avec REAL des critères d'optimisation. Un critère important à implanter pourrait être la latence bout-en-bout, c'est-à-dire la latence d'un flot de donnée. Il est possible d'exploiter plus finement les flots et modes AADL dans les théorèmes REAL, de manière à calculer ce critère. Ce travail, cependant, demande une réécriture partielle du module d'instancition de Ocarina.
L'aspect réparti de la problématique est également peu traité. Il faudrait prendre en compte le coût des communications dans le critère de latence.

L'ensemble du processus doit ensuite être entièrement automatisé - en particulier l'étape d'enrichissement du modèle. Pour cela, il faut également faire un travail de comparaison des outils d'analyse de code exécutable, et trouver des outils dont l'analyse est suffisamment fine pour répondre à nos besoins.

En ce qui concerne les fonctions d'optimisation, celles-ci peuvent être améliorées tant au niveau du modèle produit qu'au niveau du code généré. En premier lieu, l'algorithme d'ordonnancement local aux threads fusionnés devrait prendre en compte l'ordre du modèle initial, qui peut être obtenu en effectuant un tri topologique sur les tâches. En second lieu, les auto-connexions créées par les fusions successives devraient être optimisées en supprimant les mutex auxquels elles sont associées.

8.3.2 Traçabilité

La traçabilité est une exigence reconnue dans le domaine de l'aéronautique [6]. Ocarina offre une traçabilité entre le modèle AADL, l'arbre d'instances et l'arbre de la dorsale, mais pas entre ces derniers et le code généré. Un processus d'optimisation complique encore la résolution de cette tâche, en établissant une étape supplémentaire dans le processus de modélisation.

Pour garantir la traçabilité du système, il faut donc trouver un moyen d'associer chaque élément d'un modèle optimisé à l'élément dont il est issu dans le modèle original. En gardant l'implantation actuelle d'Ocarina, une telle approche n'est pas possible. Il faut donc changer l'implantation, et rendre possible la cohabitation en mémoire de plusieurs modèles et de plusieurs instances AADL, ainsi que d'établir une interface de navigation entre ces instances.

8.3.3 Vérification comportementale

Le processus peut garantir un certain nombre de propriétés sur les modèles générés (par exemple la conformité au standard ARINC653, cf. chapitre 5). Ces contraintes, cependant, doivent être exprimées sur des éléments architecturaux du modèle. L'aspect comportemental échappe assez largement à la portée d'un théorème REAL. Pour ces aspects, il peut donc être nécessaire d'avoir un modèle comportemental de l'application. Un langage d'annexe comportementale, proposé par SAE [11], répond à ce besoin. Un langage d'interrogation sur le modèle comportemental de l'application permettrait de compléter la vérification architecturale du modèle par une vérification comportementale.

Une autre approche possible est de produire un modèle formel du système (réseau de Petri et variantes colorées/temporisées, langage B ou Z), en utilisant les informations architecturales pour générer une partie de ce modèle, puis d'utiliser les méthodes associées à ces modèles formels pour exprimer des contraintes comportementales. C'est l'approche qu'a suivi l'auteur de [81], dans un travail complémentaire au notre.

8.3.4 Extension à d'autres langages de modélisation

Nous avons vu dans le chapitre 2 que les langages permettant de mettre en oeuvre notre approche devaient satisfaire certaines contraintes :

1. claire séparation des parties architecturales et fonctionnelles ;
2. visibilité des composants "systèmes" (threads, process...) ;
3. possibilité d'annotations ;
4. supports d'outils pour l'analyse des performances.

Par ailleurs, l'homogénéité de la description est un avantage dans la mesure où elle permet de contrôler simplement l'intégrité de l'architecture. Il est à noter que la multiplicité des vues d'un problème tend à augmenter la difficulté de réaliser des outils d'analyse sur le modèle, ce qui est un désavantage supplémentaire. Un manque de standardisation défavorise également l'émergence d'outils pour l'analyse. Une standardisation reconnue permet de faciliter non seulement le développement d'outils d'analyse, mais également l'adoption du langage de modélisation lui-même dans un cadre industriel.

Les deux premières conditions définissent un langage architectural. Ces langages de modélisation sont donc candidats à notre approche, éliminant les approches par composants. Les trois conditions suivantes défavorisent les langages trop spécifiquement liés à des approches ou des

outils particuliers, tels que EAST-ADL. Parmi les langages présentés dans le chapitre 2, UML/-MARTE est le meilleur candidat pour remplacer AADL, même si l'hétérogénéité du formalisme augmenterait significativement le travail à effectuer effectuer pour mettre en œuvre la transformation de modèle.

Publications

Les travaux que nous avons présentés ont donné lieu à des publications dans diverses conférences internationales ainsi que dans différents workshops. Nous présentons ci-après la liste de ces publications.

2010

"A MDE-based optimisation process for Real-Time systems" Olivier Gilles and Jérôme Hugues. Proceedings of the 13th International object/component/Service-Oriented Real-time distributed Computing (ISORC 2010). Parador of Carmona, Spain. May 2010.

"Expressing and enforcing user-defined constraints of AADL models" Olivier Gilles and Jérôme Hugues. Proceedings of the 5th UML& AADL Workshop (UML&AADL 2010). University of Oxford, UK. March 2010.

2009

"Model-Based Engineering for the Development of Partitioned Architectures" Julien Delange, Olivier Gilles, Jérôme Hugues, and Laurent Pautet. SAE AeroTech Congress & Exhibition - Avionics - Integrated Model-based System, Application and Architectures. Seattle, USA. November 2009.

"Towards Model-based optimisations of Real-Time systems, an application with the AADL" Olivier Gilles and Jérôme Hugues. 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2009). Beijing, China. August 2009. pp. 129-134.

2008

"Applying WCET analysis at architectural level" Olivier Gilles and Jérôme Hugues. Worst-Case Execution Time (WCET'08). Prague, Czech Republic. July 2008. pp. 113-122.

"Validating requirements at model-level" Olivier Gilles and Jérôme Hugues. Ingénierie Dirigée par les modèles (IDM'08). Mulhouse, France. June 2008. pp. 35-49.

A

BNF de REAL

theorem_declaration::= **theorem** [*theorem_identifier*] *range_definition* { *set_declaration* } [*required_definition*] *verification_expression* | *evaluation_expression* **end** [*theorem_identifier*] (1)

range_definition::= **foreach** *element_identifier* **in** *set_expression* **do** (2)

set_declaration::= *set_identifier affectation_operator braces_opener element_binding* | *selection_expression braces_closer* (3)

element_binding::= { *element_identifier* **in** *set_expression* } (4)

selection_expression::= *selection_expression* {*boolean_operator selection_relation*} | *generic_expression* (5)

selection_relation::= *selection_relation_identifier* ({ *selection_relation_parameter* }) (6)

selection_relation_identifier::= **Is_Subcomponent_Of** | **Is_Bound_To** | **Is_Connected_To** | **Compare_Property_Value** | **Is_Called_By** | **Is_Calling** | **Is_Accessed_By** | **Is_Connecting_To** | **Is_Accessing_To** | **Is_Feature_Of** (7)

selection_relation_parameter::= *literal* | *set_identifier* | *element_identifier* (8)

required_definition::= **requires** (*theorem_identifier* {, *theorem_identifier*}) (9)

verification_expression::= **check** (*generic_expression comparison_operator generic_expression*) (10)

evaluation_expression::= **return** ([*aggregation_function_identifier* (] *generic_expression*)[)] (11)

generic_expression::= *literal* | *verification_function_call* | expr_expression | (*generic_expression*) | *generic_expression generic_operator generic_expression* | *ternary_expression* (12)

expr_expression::= **expr** (*set_identifier* , *variable_identifier* , generic_expression) (13)

set_expression::= *set_identifier* | *set_expression set_operator set_expression* (14)

ternary_expression::= **if** *generic_expression* **then** *generic_expression* **else** *generic_expression* (15)

verification_function_call::= *verification_function_identifier* ({ *verification_function_parameter* }) (16)

verification_function_parameter::= *literal* | *set_identifier* | *element_identifier* | *verification_function_identifier* (17)

variable_declaration::= **var** *variable_identifier affectation_operator variable_definition* (18)

variable_definition::= *variable_declaration_subtheorem* | *generic_expression* (19)

variable_declaration_subtheorem::= **compute** *theorem_identifier* ({ *verification_function_parameter*) } (20)

aggregation_function_identifier ::= **MSum MMax MProd MMin** (21)

verification_function_identifier::= **Cardinal** | **Sum** | **Max** | **Min** | **Product** | **GCD** | **LCM** | **Get_Property_Value** | **Property** | **Property_Exists** | **Exists** | **System** | **First** | **Last** | **Head** | **Queue** | **List** | **Size** (22)

set_operator::= + | * | / (23)

theorem_identifier::= '⟨'string'⟩' (24)

set_identifier::= '⟨'string'⟩' (25)

element_identifier::= '⟨'string'⟩' (26)

variable_identifier::= '⟨'string'⟩' (27)

generic_operator::= *comparison_operator* | *boolean_operator* | + | - | * | / | ** (28)

boolean_operator::= **and** [**not**] | **or** (29)

comparison_operator::= > | >= | <= | <| = | <> (30)

literal::= *boolean_literal* | *string_literal* | *numeric_literal* (31)

affectation_operator::= := (32)

braces_opener::= { (33)

braces_closer::= } (34)

Bibliographie

[1] http://www.aadl.info/.

[2] http://www.eclipse.org/.

[3] http://www.omg.org.

[4] http://www.rtems.com.

[5] http://www.windriver.com/.

[6] *Software considerations in airborne systems and equipement certification*, december 1992.

[7] *CORBA Component Model Specification v4.0*, 1997.

[8] *Lightweight CCM specification*, 2005.

[9] *Architecture Analysis and Design Language (AADL)*, 2006.

[10] *Deployment and Configuration of Component-based Distributed Applications Specification*, 2006.

[11] *Annex X Behavior Annex*, december 2009.

[12] *Architecture Analysis and Design Language v2 (AADLv2)*, 2009.

[13] ISO/IEC 13568. Z formal specification notation — syntax, type system and semantics, 2002.

[14] T. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback performance control in software services. *IEEE Control Systems*, 23(3), 2003.

[15] A. Aleti, S. Bjornander, L. Grunske, and I. Meedeniya. Archeopterix : An extendable tool for architecture optimization of aadl models. In *Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, 2009.

[16] C. André. Representation and analysis of reactive behaviors : A synchronous approach. In *Computational Engineering in Systems Applications (CESA)*, 1996.

[17] ATESST Consortium. *EAST ADL 2.0 Specification*, 2008.

[18] ATMEL. *TSC695 SPARC V7 Processor (ERC32) Development Tools*, 2005.

[19] AUTOSAR. *Main Requirements V2.1.0*, 2009.

[20] K. Balasubramanian and D. C. Schmidt. Physical Assembly Mapper : A Model-driven Optimization Tool for QoS-enabled Component Middleware. In *Proceedings of RTAS'08*, 2008.

[21] T. Barris, R. Ameur-Boulifa, A. Cansado, L. Henrio, and E. Madelaine. Behavioural models for distributed fractal components. *Annales des Télécommunications*, 64, 2009.

[22] D. E. Bell and L. J. LaPadula. Secure computer system : Unified exposition and multics interpretation. Technical report, The MITRE Corporation, 1976.

[23] A. Benoit. *Scheduling Pipelined Applications : Models, Algorithms and Complexity*. Habilitation à diriger des recherches, École Normale Supérieure de Lyon, 2009.

[24] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1991.

[25] G. Berry and G. Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of computer programming*, (2), 1992.

[26] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE CORP BEDFORD MA, 1977.

[27] B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21, 1988.

[28] C. Boettcher, R. DeLong, J. Rushby, and W. Sifre. The mils component integration approach to secure information sharing. In *Proceedings of the 27th Digital Avionics Systems Conference*, 2008.

[29] E. Borde, G. Haik, V. Watine, and L. Pautet. Really hard time developping hard real time. In *Workshop on Control Architecture of Robots 2007 (CAR'07)*, 2007.

[30] Víctor Braberman, Diego Garbervetsky, and Sergio Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5) :31–58, jun 2006.

[31] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model. Technical report, ObjectWeb Consortium, 2003.

[32] W. Burns. Arinc 653 and why is it important for a safety-critical rtos. *Boards & Solutions*, pages 16–18, April 2004.

[33] A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *Informs Journal on Computing*, 11 :125–137, 1998.

[34] A. Caprara, D. Pisinger, and P. Toth. Exact solution of the quadratic knapsack problem. *Informs Journal on Computing*, 11 :125–137, 1999.

[35] D. Cornhill, L. Sha, and J. P. Leboczky. Limitations of ada for real-time scheduling. In *Proceedings of the First international Workshop on Real-time Ada Issues (IRTAW'87)*, 1987.

[36] R.I. Davis and A. Burns. Response time upper bounds for fixed priority real-time systems. In *Real-Time Systems Symposium, 2008*, pages 407–418, 30 2008-Dec. 3 2008.

[37] D. de Niz and R. Rajkumar. Partitioning bin-packing algorithms for distributed real-time systems. *International Journal of Embedded Systems*, 2006.

[38] J. Delange, O. Gilles, J. Hugues, and L. Pautet. Model-based engeineering for the development of arinc653 architectures. In *AEROTECH'09*, November 2009.

[39] J. Delange, O. Gilles, J. Hugues, and L. Pautet. Model-based engineering for the development of partitioned architectures. In *SAE AeroTech Congress & Exhibition - Avionics - Integrated Model-based System, Application and Architectures*, 2009.

[40] J. Delange, L. Pautet, and F. Kordon. Code generation strategies for partitioned systems. In IEEE Computer Society, editor, *29th IEEE Real-Time Systems Symposium (RTSS'08) Work In Progress*, December 2008.

[41] S. Demathieu, F. Thomas, C. Andre, S. Gerard, and F. Terrier. First experiments using the uml profile for marte. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[42] I. Demeure and C. Bonnet. *Introduction aux systèmes temps-réel.* Hermes, 1999.

[43] B. Demuth and C. Wilke. Model and object verification by using dresden OCL. In *Proceedings of the Russian-German Workshop "Innovation Information Technologies : Theory and Practice"*, 2009.

[44] B. Dobbing, A. Burns, and T. Vardanega. Guide for the use of the of the Ravenscar Profile in High Integrity Systems. Technical report, University of York, 2003.

[45] M. Ducassé and L. Rozé. Proof obligations of the b formal method : Local proofs ensure global consistency. In *Logic Programming Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Selected Papers*, 2000.

[46] S. A. Edwards, V. Kapadiab, and M. Halasb. Compiling esterel into static discrete-event code. In *Proceedings of the Third International Workshop on Synchronous Languages, Applications, and Programs*, 2004.

[47] S. A. Edwars and J. Zeng. Code generation in the columbia esterel compiler. *EURASIP Journal on Embedded Systems*, 2007.

[48] Airlines Electronic Engineering. Avionics application software standard interface. Technical report, Aeronautical Radio, INC, 1997.

[49] J. L. Lions et al. ARIANE V, FLight 501 Failure. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, 1996.

[50] R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. *SIGOPS Oper. Syst. Rev.*, 11(5) :57–65, 1977.

[51] P. H. Feiler and J. Hansson. Flow latency analysis with the architecture analysis and design language. Technical Report CMU/SEI-2007-TN-010, Carnegie Mellon University - Software Ingineering Institute, 2007.

[52] O. Gilles and J. Hugues. Applying WCET analysis at architectural level. In *Worst-Case Execution Time (WCET'08)*, pages 113–122, Praga, Czech Republic, jul 2008.

[53] O. Gilles and J. Hugues. Towards Model-based optimisations of Real-Time systems, an application with the AADL. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, Beijing, China, August 2009.

[54] O. Gilles and J. Hugues. A MDE-based optimisation process for Real-Time systems. In *Proceedings of the 13th International object/component/Service-Oriented Real-time distributed Computing (ISORC 2010)*, Parador of Carmona, Spain, may 2010.

[55] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system c programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, pages 287–297, feb 2005.

[56] I. Hamid. *Automatic Code Generation and Verification of Hard Real-time Systems*. PhD thesis, École Normale Supérieure de Télécommunications, may 2008.

[57] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7) :1038–1054, July 2003.

[58] N. Holsti. Computing time as a program variable : A way around infeasible paths. In *8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2008.

[59] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4) :1–25, jul 2008.

[60] Jérôme Hugues. *Architecture et Services des Intergiciels Temps Réel.* Thèse de doctorat, École nationale supérieure des télécommunications, September 2005.

[61] Jérôme Hugues, Bechir Zalila, Laurent Pautet, and Fabrice Kordon. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*, pages 106–112, Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.

[62] ISO. *Information Technology – Programming Languages – Guide for the use of the Ada programming language in high integrity systems.* ISO, 2000.

[63] J.-R.Abrial. *The B book - Assigning Programs to meanings.* Cambridge University Press, 1996.

[64] B. A. Julstrom. Greedy, genetic, and greedy genetic algorithms for the quadratic knapsack problem. In *GECCO'05*, 2005.

[65] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springers, 2004.

[66] L. Kinnan, J. Wlad, and P. Rogers. Porting applications to an arinc 653 compliant ima platform using vxworks as an example. *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, 2 :10.B.1–10.1–8 Vol.2, 24-28 Oct. 2004.

[67] B. Lewis, E. Colbert, and S. Vestal. Developing evolvable, embedded, time-critical systems with metah. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, 2000.

[68] C. D. Locke, D. R. Vogel, and J. B. Goodenough. Generic avionic software specification. Technical report, Carnegie Mellon University, Pittsburg, Pennsylvania, USA, 1990.

[69] C. D. Locke, D. R. Vogel, and T. J. Mesler. Building a predictable avionic plateform in ada : A case study. In *IEEE Real-Time Systems Symposium*, pages 181–189, 1991.

[70] C. Lu, X. Wang, and X. D. Koutsoukos. Feedback utilization control in distributed real-time systems with end-to-end tasks. *IEEE Transactions on Parallel and Distributed Systems*, 16(6), 2005.

[71] F. Mallet. Clock constraint specification language. Technical report, INRIA, 2008.

[72] P. L. Martinez, J. M. Drake, P. Pacheco, and J. L. Medina. Ada-ccm : Component-based technology for distributed real-time systems. In *11th International Symposium on Component-based Software Engineering (CBSE 2008)*, 2008.

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

[74] M. Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Trans. Embed. Comput. Syst.*, 7(3) :1–32, 2008.

[75] D. Niz. Diagrams and languages for model-based software engineering of embedded systems : Uml and aadl. Technical report, SEI, 2007.

[76] D. Niz and P.H. Feiler. On resource allocation in architectural models. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[77] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification*, 2006. http://www.omg.org/spec/MOF/2.0/.

[78] Object Management Group (OMG). *OMG SysML Specification*, 2006.

[79] N. Pernet. *Implantation distribuée temps réel de programmes conditionnés à l'aide d'ordonnancements mixtes hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques.* PhD thesis, Université de Paris 6, Spécialité Informatique, 07/07/2006.

[80] S. Popinchalk, J. Glass, R. Shenoy, and R. Aberg. Working in teams : Modeling and control design within a single software environment. In *Proceedings of AIAA'07*, 2007.

[81] X. Renault. *Mise en oeuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systèmes embarqués temps-réel répartis.* PhD thesis, Université Pierre et Marie Curie, Spécialité Informatique, 2009.

[82] J. Rushby. Separation and integration in mils (the mils constitution). Technical report, SRI International, 2008.

[83] F. Singhoff, J. Legrand, L. Nana, and L. Marc. Cheddar : a flexible real time scheduling framework. In *ACM SIGAda Ada Letters*, New York, USA, December 2004. ACM Press.

[84] C.-J. Sjöstedt, D. Chen, P. Cuenot, P. Frey, R. Johansson, H. Lönn, D. Servat, and M. Törngren. Developing dependable automotive embedded systems using the east-adl ; representing continuous time systems in sysml. In *1st Int. Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'2007)*, 2007.

[85] Y. Sorel. SynDEx : System-level cad software for optimizing distributed real-time embedded systems. *ERCIM News*, (59), 2004.

[86] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *5th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[87] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhlem, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, 2003.

[88] Adam Torgerson. Automatic thread stack management for resource-constrained sensor operating systems. 2005.

[89] X. Wang, Y. Chen, C. Lu, and X. D. Koutsoukos. Towards controllable distributed real-time systems with feasible utilization control. *IEEE Transactions on Computers*, 58(8), 2009.

[90] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3) :1–53, 2008.

[91] B. Zalila. *Configuration et déploiement d'applications temps-réel réparties embarquées à l'aide d'un langage de description d'architecture.* PhD thesis, École Nationale Supérieure des Télécommunications, nov 2008.

[92] B. Zalila, L. Pautet, and J. Hugues. Towards automatic middleware generation. In *Proceedings of the 11th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'08)*, Orlando, Florida, USA, May 2008. IEEE Computer Society Press.

[93] J. Zamorano and J. F. Ruiz. Gnat/ork : An open cross-developpment environment for embedded ravenscar. In *Reliable Software Technologies – Ada-Europe 2000*, 2002.

Résumé

Une démarche classique d'ingénierie dirigée par les modèles (IDM) consiste à modéliser un problème, puis à générer le code source associé à partir de ce modèle. Cette approche, qui a été étendue succès aux systèmes temps-réel, réduit significativement les erreurs.

Le modèle développé suit généralement une approche fonctionnelle du problème ; celle-ci est cependant rarement optimale en terme de consommation de ressources. Pour les systèmes temps-réel, cette limitation est acceptable : ce n'est plus le cas dans le contexte de systèmes temps-réels embarqués (TRE).

Cette thèse propose d'associer l'approche IDM à un processus d'optimisation basé sur le modèle du syst 'eme, pour pouvoir appliquer cette approche aux systèmes TRE. Pour cela, nous proposons d'utiliser ensemble trois solutions : d'une part, AADL, un langage de modélisation architecturale, qui permet de spécifier la composante non-fonctionnelle de l'application. Ensuite, REAL, un langage d'expression de contrainte, qui permet d'exprimer des contraintes sur le modèle. Finalement, un processus d'optimisation, qui permet de transformer un modèle "naïf" en modèle répondant aux performances requises, en se basant sur des heurstiques gloutonnes.

Nous montrons comment cette approche permet d'automatiser le processus de développement, en limitant le rôle de l'architecte à la traduction des contraintes exprimées par le cahier des charges et à la conception d'un modèle naïf du problème.

Mots-clés: systèmes temps-réel, systèmes embarqués, optimisation, modélisation, évaluation, transformation de modèle, langage d'expression de contraintes, génération de code.

Abstract

A common usage in model-driven approach (MDA) consist in modeling a problem, then generate the associated source code. This developping process had been successfully applied to real-time systems and reduces significantly the number of failure.

The designed model generaly follows a functional cut of the problem which is usualy not optimal in terms of resources consumption. Although real-time systems can bear this limitation, it is not the case for embedded real-time (ERT) systems.

This thesis advocates for associating the benefits of MDA with an optimization process of the system model, in order to applied it to ERT systems. To do so, we use togeither three solutions : first, AADL, an language for architectural modeling allowing to specify non-fonctional aspects of the application. Then, REAL, a language for expressing constraints on the AADL models. Finally, an optimization process which allows to turn a "naïve", fonctionnaly-cut model into a model actually meeting the resources constraints, using greedy heuristics.

We show how this approach allows to automate all the developping process, limiting the designer role to translating the constraints expressed in the initial specification and modeling the corresponding naïve model.

Keywords: real-time systems, embedded systems, optimization, modéling, évaluation, model transformation, constraint expression language, code generation.