Modélisation et contrôle formel de la reconfiguration – Application aux systèmes embarqués dynamiquement reconfigurables

Guillet Sébastien

To cite this version:

Guillet Sébastien. Modélisation et contrôle formel de la reconfiguration – Application aux systèmes embarqués dynamiquement reconfigurables. Génie logiciel [cs.SE]. Université de Bretagne Sud, 2012. Français. NNT : . tel-00879731

HAL Id: tel-00879731
https://theses.hal.science/tel-00879731

Submitted on 4 Nov 2013

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.

THÈSE / UNIVERSITÉ DE BRETAGNE SUD
Lorient
Sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE-SUD
Mention : STIC
Écloe Doctorale SICMA

présentée par

Sébastien GUILLET

Lab-STICC

Modélisation et contrôle formel de la reconfiguration

Application aux systèmes embarqués dynamiquement reconfigurables

Thèse soutenue le 5 décembre 2012
devant la commission d'examen composée de :

Bertrand GRANADO
Professeur des Universités à l'UPMC / Examinateur

Robert DE SIMONE
Directeur de recherche à INRIA Sophia-Antipolis / Rapporteur

Sébastien PILLEMENT
Professeur des Universités à Polytech Nantes / Rapporteur

Guy GOGNIAT
Professeur des Universités à l'UBS / Directeur de thèse

Éric RUTTEN
Chargé de recherche au LIG/INRIA Rhône-Alpes / Co-directeur de thèse

Florent DE LAMOTTE
Maître de Conférence à l'ENSIBS / Co-directeur de thèse

Table des matières

Introduction générale **15**
1 Enjeux de la dynamicité des systèmes embarqués 15
2 Contexte de projet 16
3 Exemple informel 16
4 Plan 19

I État de l'art **23**

1 Architectures reconfigurables **25**
1.1 Introduction 25
1.2 Principes et compromis 27
1.2.1 Performance 27
1.2.2 Flexibilité 28
1.2.3 Classification 29
1.3 Systèmes embarqués adaptatifs 30
1.3.1 Systèmes sur puce reconfigurables 30
1.3.2 Field Programmable Gate Array 32
1.3.3 Reconfiguration Dynamique et Partielle (RDP) 33
1.4 Études et travaux exploitant les architectures reconfigurables 34
1.4.1 Prototypage 34
1.4.2 Mise à jour dynamique 35
1.4.3 Tolérance aux fautes 35
1.4.4 Contrôle thermique 36
1.4.5 Optimisation de la qualité de service 36
1.5 Conclusion 37

2 Co-conception de systèmes sur puces 39
2.1 Introduction 39
2.2 Modèles, méthodes et outils 41
2.2.1 High Level Synthesis 41
2.2.2 Domain Specific Languages 42
2.2.3 Model Driven Engineering 43
2.3 MDE pour la conception de SoC 49
2.3.1 Profils UML 49
2.3.2 UML/MARTE en particulier 50
2.3.3 UML/MARTE pour la modélisation des SoC reconfigurables . . 53
2.4 Conclusion 56

3 Contrôle de la reconfiguration 57
3.1 Introduction 57
3.2 Notion de conception sûre 58
3.2.1 Approches par test 58
3.2.2 Approches par méthodes formelles 61
3.3 Spécification du contrôle 66
3.3.1 Systèmes réactifs synchrones 66
3.3.2 Synchronisme 68
3.3.3 Langages synchrones 69
3.3.4 D'autres approches de spécification du contrôle 73
3.4 Conclusion 74

II Contribution et méthodologie 77

4 Modélisation du contrôle dans MARTE 79
4.1 Introduction 79
4.2 Sémantique des élements standards 80
4.2.1 Modélisation de l'application 81
4.2.2 Modélisation de l'architecture 88
4.2.3 Modélisation de l'allocation 88
4.2.4 Ajout de propriétés et contraintes 90
4.3 Extension du profil MARTE 93
4.3.1 Composition générique des propriétés non fonctionnelles 94

4.3.2 Modélisation d'un contrôleur générique . . . 96
4.4 Conclusion . . . 100

5 Transformation du contrôle vers une représentation synchrone 103
5.1 Introduction . . . 103
5.2 Méta-modèle intermédiaire dédié au contrôle . . . 104
5.2.1 Automates de modes hiérarchiques . . . 104
5.2.2 Combinaison des propriétés non fonctionnelles . . . 106
5.2.3 Propriétés du contrôleur . . . 108
5.3 Implémentation exécutable du contrôle . . . 109
5.3.1 Challenge de l'implémentation maximalement permissive . . . 109
5.3.2 Transformation vers BZR et exploitation de la synthèse de contrôleur . . . 114
5.4 Problème du non déterminisme de la représentation en automates . . . 117
5.4.1 Exemple . . . 119
5.4.2 Différence sémantique entre les ModeBehavior et les automates en BZR . . . 122
5.4.3 Exploitation du non-déterminisme pour la décision en ligne de reconfiguration . . . 123
5.5 Conclusion . . . 123

6 Combinaison du contrôle discret et de l'optimisation sous contrôle 125
6.1 Introduction . . . 125
6.2 Notion de décision après contrôle . . . 126
6.2.1 Nécessité de la décision en ligne . . . 126
6.2.2 Représentation équationnelle non déterministe des automates . 127
6.2.3 Problématique de la prise en compte du mécanisme de décision dans la spécification de contrôle . . . 128
6.3 Formalisation du problème et adaptation des transformations . . . 128
6.3.1 Définitions . . . 128
6.3.2 Adaptation des transformations . . . 129
6.4 Exemple de décision générique . . . 132
6.4.1 Frontière de Pareto dynamique . . . 134
6.4.2 Régulation par contrôle PI . . . 135
6.4.3 Élection d'une configuration . . . 139
6.5 Conclusion . . . 140

III Validation, développement et évaluation 143

7 Définition d'une architecture de démonstration 145
7.1 Introduction 145
7.2 Définition des besoins 146
7.2.1 Capacité de reconfiguration 147
7.2.2 Méthode d'exécution 147
7.3 Spécifications de l'architecture cible 148
7.3.1 Établissement d'un pipeline générique de traitement vidéo . . . 149
7.3.2 Adaptativité du pipeline 150
7.4 Spécification du type d'application ciblée 152
7.4.1 Démonstrateur pour la synthèse de contrôleur 152
7.4.2 Application de suivi d'objets 153
7.5 Matériel considéré 154
7.6 Conclusion 154

8 Modélisation du cas d'étude 157
8.1 Introduction 157
8.2 Propriétés de l'exemple d'application 158
8.2.1 Contraintes 158
8.2.2 Besoins 159
8.3 Modélisation 160
8.3.1 Application 160
8.3.2 Architecture 167
8.3.3 Allocation 168
8.4 Partie manuelle 169
8.5 Conclusion 169

9 Automatisation du flot 171
9.1 Introduction 172
9.2 Génération de la partie synchrone 173
9.2.1 Interface du nœud de contrôle MyController 173
9.2.2 Hypothèse 175
9.2.3 Objectif de contrôle 177
9.2.4 Définition de l'accessibilité des états 177
9.2.5 Définition de l'accessibilité des configurations 178

9.2.6 Automates réagissant d'après la décision . . . 178
9.2.7 Calcul et attribution des combinaisons de poids . . . 178
9.2.8 Compilation et synthèse . . . 180
9.3 Génération du support logiciel pour l'implémentation de la décision . . 181
9.3.1 Génération de l'interface de décision . . . 181
9.3.2 Accès aux métriques des configurations . . . 183
9.4 Support pour la simulation . . . 186
9.4.1 Communication vers le contrôleur . . . 187
9.4.2 Visualisation de l'exécution du contrôle . . . 187
9.5 Exécution du démonstrateur . . . 189
9.5.1 Simulation . . . 189
9.5.2 Chaîne de simulation . . . 193
9.5.3 Intégration dans la plateforme . . . 198
9.6 Conclusion . . . 200

Conclusion générale **201**
1 Bilan . . . 201
1.1 Définition d'une méthodologie reposant sur UML/MARTE . . . 201
1.2 Spécification de transformations . . . 202
1.3 Intégration du contrôle discret statique et sûr à la décision pour optimisation . . . 202
1.4 Proposition d'une plateforme reconfigurable . . . 203
1.5 Mise en place d'une chaîne de simulation . . . 203
1.6 Comparaison par rapport aux travaux fondateurs . . . 204
2 Perspectives . . . 205
2.1 Perspectives sur la méthodologie proposée . . . 205
2.2 Perspectives d'intégration dans le projet FAMOUS . . . 212

Bibliographie **215**

Table des figures

1 Graphe de tâches, dont deux sont reconfigurables. 17
2 Modèle comportemental sous forme de machines à états, permettant de désigner les implémentations à instancier. 17
3 La construction d'un modèle comportemental capable de valuer certains événements désignés afin de respecter des contraintes pour toute exécution est un problème difficile. 18

1.1 Exemple de système sur puce, constitué d'un processeur ARM et de diverses interfaces de communication. 31
1.2 Principe d'exécution générique avec contrôleur de reconfiguration 38

2.1 Niveaux de modélisation du MDE et exemples de modèles 46
2.2 Architecture globale du profil MARTE 51
2.3 Niveau de modélisation détaillé dans MOPCOM 55
2.4 Transformations de modèles dans MOPCOM 55

3.1 Modèle en cascade 58
3.2 Cycle en V 59
3.3 Représentation visuelle d'un système réactif qui, à chaque activation par l'environnement, lit des entrées ($\mathcal{I}$), les traite en mettant son état à jour, et produit un résultat ($\mathcal{O}$). 68
3.4 Système de contrôle-commande d'un avion 68
3.5 Traduction du programme HEPTAGON/BZR (`y = x -> pre(y) + x;`) en réseau de Kahn. 71

4.1 Principe de modélisation d'un graphe de tâches en MARTE 82
4.2 Modélisation d'un contrôleur reponsable du flot de contrôle et de l'ordonnancement de tâches dans MOPCOM 83
4.3 Définition d'un Mode Switch Component (MSC) dans Gaspard, montrant une liaison de type "un mode = une implémentation" 84

4.4 Modélisation d'un composant reconfigurable dans Gaspard 84
4.5 Comportement d'une tâche générique, acceptant (s'il y a lieu) un ordre de reconfiguration et le diffusant à ses tâches immédiatement connectées dans le graphe 86
4.6 Définition de ModeBehaviors et de Configurations MARTE associées 87
4.7 Vue en composant des Configurations MARTE, définissant les aspects de configurations à mettre en œuvre lorsque leur mode associé est actif 87
4.8 Modélisation de l'allocation de la tâche T1 sur un processeur MicroBlaze (élément de modèle devant être présent dans le PM) 89
4.9 Définition d'une hiérarchie de tâches implémentant T2, et adaptation des Configurations MARTE pour refléter leur mise en œuvre concrète suivant l'activation de leur mode respectif 89
4.10 Allocation de chaque nouvelle tâche définie dans la hiérarchie vers leur élément de plateforme respectif 90
4.11 Package MARTE NFP_Declaration, dédié à la définition de types et valeurs de propriétés non fonctionnelles 91
4.12 Extrait de la librairie de types de NFP de base dans MARTE 92
4.13 Définition de contraintes temporelles à un composant ; "contract" signifie que la contrainte est respectée par le composant lui-même, "required" signifie que la contrainte est supposée être respectée par l'environnement du composant 92
4.14 Association de valeurs de propriétés non fonctionelles à un composant . . . 93
4.15 Amélioration du profil MARTE pour le support des combinaisons de valeurs de NFP 95
4.16 Définition de deux ModeBehaviors à synchroniser 96
4.17 Synchronisation à l'aide d'un RtUnit et spécification de contraintes d'exécution 96
4.18 Définition d'un élément interne au RtUnit de contrôle, dédié à la résolution des valeurs d'événements internes 98
4.19 Extension de MARTE pour compléter le support en terme de spécification de contrôle 99
4.20 Utilisation du nouveau stéréotype pour l'identification de ce RtUnit en tant qu'élément de contrôle, identification du resolver à synthétiser, et déclaration des types de NfpMeasures à combiner 100

5.1 Extrait du profil MARTE dédié à la spécification de comportement modal 105
5.2 Extrait du métamodèle intermédiaire pour la représentation comportementale 105

5.3 Extrait du métamodèle intermédiaire pour la représentation des configurations . . . 106
5.4 Principe de définition de types dans le métamodèle intermédiaire . . . 107
5.5 Principe de définition des opérations sur poids dans le métamodèle intermédiaire . . . 108
5.6 Principe de définition des contrôleurs dans le métamodèle intermédiaire . . 108
5.7 Exemple de système à contrôler, représenté par un LTS . . . 111
5.8 Extraction de la région maximale d'état autorisés dans le LTS d'exemple . 113

6.1 Principe d'intégration contrôle/décision . . . 126
6.2 Modèle en boucle fermée d'adaptation d'un système, avec un régulateur PI 135

7.1 Pipeline de traitement vidéo . . . 149
7.2 Contrôle bas-niveau de la reconfiguration, effectué via un MicroBlaze . . . 150
7.3 Ajout de capacités de traitements d'images exploitant des co-processeurs . 152

8.1 Graphe de tâches représentant l'exécution en séquence de *Resolution* et *Filter* 161
8.2 Représentation des liaisons tâches/implémentations via des Configurations MARTE . . . 161
8.3 Définition du comportement via des ModeBehaviors . . . 161
8.4 Synchronisation des ModeBehaviors via un RtUnit dédié au contrôle . . . 162
8.5 Spécification des types de NfpMeasures *power* et *qos* . . . 165
8.6 Attribution de valeurs de NfpMeasures aux Configurations MARTE . . . 166
8.7 Prise en compte des NfpMeasures pour ce contrôleur (déclaration de types, contrainte sur poids, décision) . . . 166
8.8 Représentation du pipeline de traitement vidéo pour cet exemple de démonstration . . . 168
8.9 Allocation des tâches du modèle d'application aux supposés éléments reconfigurables du modèle d'architecture . . . 168

9.1 Flot de conception, transformation, simulation et intégration pour la partie contrôle . . . 172
9.2 Calcul de la frontière de Pareto dynamique. . . . 186
9.3 Premier pas de contrôle/décision. . . . 189
9.4 Deuxième pas de contrôle/décision. . . . 191
9.5 Troisième pas de contrôle/décision. . . . 192
9.6 Envoi d'événements, contrôle et décision pour le premier pas. . . . 194
9.7 Envoi d'événements, contrôle et décision entre deux configurations pour le deuxième pas. . . . 194

9.8 Envoi d'événements, contrôle et décision pour le troisième pas. 195
9.9 Résultat de simulation du contrôleur dans Sim2Chro. 197
9.10 Résultat de simulation du contrôleur dans GTKWave. 197
9.11 Initialisation du système, démarrage en configuration {Medium ;Color}. . . 198
9.12 Premier pas, reconfiguration vers {High ;Color}. 199
9.13 Deuxième pas, {High ;Bw} interdite, choix de reconfiguration vers {Low ;Bw}.199
9.14 Troisième pas, {High ;Bw} interdite, le système reste en {Low ;Bw}. 199

1 Méthode systématique de modélisation permettant ensuite de simuler des coûts de transition. 207
2 Détails des configurations intermédiaires possibles d'une reconfiguration en pipeline. 210

Liste des Algorithmes

1 Exécution d'un système transformationnel. 67
2 Exécution d'un système interactif. 67
3 Exécution d'un système reactif. 68
4 Exécution d'un système reactif sous contrôle. 72

5 Extraction de la région d'états permis d'un système donné 112
6 Exemple de déclaration de nœud avec contrat 114
7 Correspondance générique en BZR de l'interface d'un contrôleur 115
8 Déclaration générique d'automates et de variables internes liées à l'activation d'états 116
9 Valuation des poids v_f de la configuration active en fonction des valeurs v_{f_i} des poids des configurations. 118
10 Exécution d'un système réactif sous contrôle maximalement permissif avec méthode de valuation complémentaire à bas d'*oracles* 120

11 Représentation vectorielle du treillis de configurations 136
12 Construction dynamique de la frontière de pareto d'un ensemble de configurations potentielles 137

13 Structure du programme BZR généré d'après la spécification RecoMARTE du démonstrateur. 174
14 Spécification de l'hypothèse comportementale de l'environnement (et donc de la décision). 176
15 Spécification de l'objectif de contrôle. 177
16 Définition de l'accessibilité des états. 177
17 Définition de l'accessibilité des configurations. 178
18 Automates d'origine, réagissant maintenant d'après les événements en provenance du système de décision. 179
19 Attribution des poids aux configurations et définition de l'équation assignant le poids courant aux variables appropriées (*power* et *qos*). 180

20 Génération de l'interface de décision, avec mécanisme de décision par défaut à adapter. 182

Introduction générale

1 Enjeux de la dynamicité des systèmes embarqués

Les systèmes embarqués ont continuellement, et de façon croissante, besoin de disposer d'une certaine flexibilité de leurs fonctionnalités pendant leur exécution : le changement des besoins de l'utilisateur, l'adaptation dynamique de fonctionnalités selon le contexte d'exécution, l'évolution de protocoles et standards de communication, etc. sont autant de raisons motivant la recherche pour la conception de systèmes embarqués dynamiquement reconfigurables, ayant la capacité de s'adapter à un environnement changeant.

Un des challenges à relever et identifié dans [28] est celui de l'adoption à une échelle large du support de la reconfiguration dynamique dans les applications embarquées de demain, support pour l'instant très pénalisé par la faible présence de ce paradigme dans les systèmes embarqués actuels. En conséquence, les moyens mis à disposition des concepteurs sont à un stade préliminaire, et la conception ciblant et exploitant les systèmes reconfigurables en général souffre d'un fossé de productivité.

C'est pour répondre à ce challenge que nous nous posons comme objectif de mettre en place une méthodologie permettant de simplifier la spécification de systèmes embarqués dont les plateformes peuvent inclure, entre autre, des processeurs et accélérateurs capables d'être reconfigurés. Cette méthodologie cible en particularité les FPGA, technologie majeure des architectures reconfigurables, spécialement ceux disposant de la technologie de *Reconfiguration Dynamique et Partielle*.

Cet objectif sera réalisé dans le cadre d'un projet de recherche qui se décompose en trois contributions principales :

1. la définition de moyens de modélisation à haut niveau d'abstraction incluant la notion de reconfiguration ; le profil UML MARTE (qui sera présenté dans la suite) est pressenti pour le support de modélisation ;
2. le support de la reconfiguration dynamique et partielle de la manière la plus transparente possible, en permettant au concepteur de faire abstraction de détails de bas

niveau (assistance à la diffusion d'une configuration impactant plusieurs tâches d'un système en cours d'exécution) ;

3. la mise en place d'un cadre formel pour l'analyse et la vérification de reconfiguration.

2 Contexte de projet

La recherche présentée dans ce manuscrit s'est effectuée au sein d'un projet ANR (Agence Nationale de la Recherche), du nom de FAMOUS (**FA**st **M**odeling and Design Fl**O**w for Dynamically Reconfig**U**rable **S**ystems), visant à introduire une méthodologie complète de modélisation pour la spécification et la conception de systèmes embarqués dynamiquement reconfigurables. Elle s'est principalement déroulée au Lab-STICC à l'Université de Bretagne-Sud (Lorient), ainsi qu'à l'INRIA Rhône-Alpes (Grenoble), en collaboration avec l'INRIA Lille – Nord Europe, le laboratoire LE2I (Dijon), et l'éditeur de logiciel Sodius (Nantes). Ces entités sont les cinq acteurs partenaires du projet, qui se répartissent les responsabilités dans la réalisation de ses contributions présentées précédemment.

Les travaux que nous présentons ici, menés de Novembre 2009 à Octobre 2012, ont pour but de décrire un aspect particulier du projet FAMOUS, concernant la modélisation du contrôle formel de la reconfiguration dynamique. Il s'agit donc ici d'apporter essentiellement une contribution à la troisième contribution concernant le contrôle formel de la reconfiguration, ainsi qu'à la première pour ce qui traite de la modélisation de l'aspect contrôle.

3 Exemple informel

Considérons un graphe de tâches décrivant une application, tel que celui montré dans la figure 1, disposant de quatre tâches (T1, T2, T3 et T4) communiquant entre elles leurs résultats d'exécution. L'exécution d'un tel graphe requiert généralement un modèle de calcul approprié qu'il n'est pas nécessaire d'aborder pour le moment. Parmi ces tâches, deux d'entre elles (T2 et T4) possèdent plusieurs implémentations, c'est-à-dire plusieurs façons exclusives de traiter une même interface d'entrées/sorties ; ces deux tâches possèdent respectivement trois et deux implémentations.

Les tâches T2 et T4 sont dites *reconfigurables*, et leur exécution nécessite d'être couplée à un modèle dit *comportemental* dont le rôle est d'indiquer comment passer d'une implémentation à une autre. Un tel modèle est représenté sous forme de machines à états informelles dans la figure 2 : sous réserve de disposer d'une sémantique appropriée, admettons ici que les états préfixés par "E2" et "E4" soient connectés respectivement aux

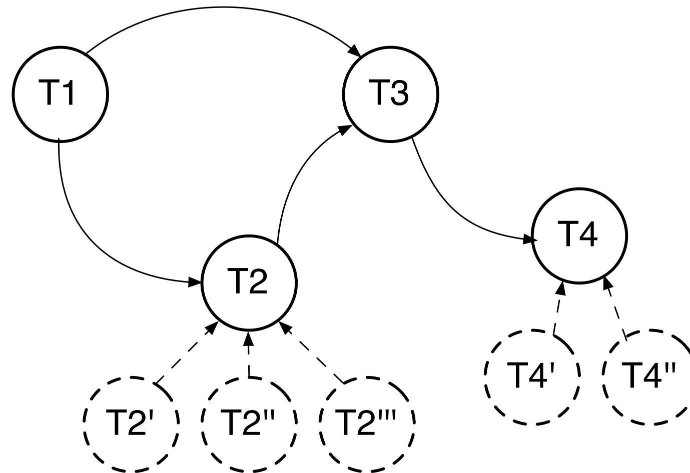


FIGURE 1 – Graphe de tâches, dont deux sont reconfigurables.

implémentations de T2 et T4, l'activation d'un état donné indiquant alors l'instanciation de l'implémentation correspondante sur sa tâche. Les gardes de ces machines à états, représentées symboliquement et préfixées ici par "e", désignent chacune une équation booléenne reposant sur des événements dont les valeurs sont fournies au système (capteurs, entrées utilisateur, etc.).

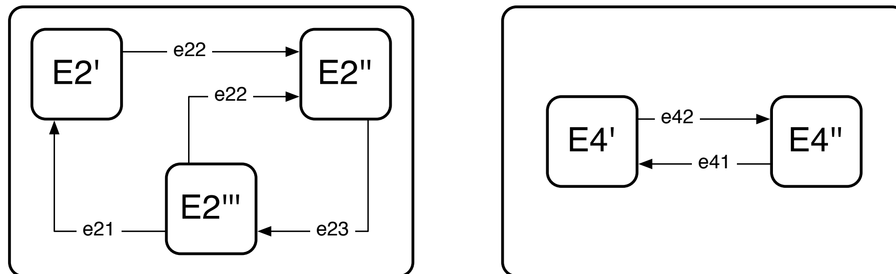


FIGURE 2 – Modèle comportemental sous forme de machines à états, permettant de désigner les implémentations à instancier.

On considère donc des instants d'exécution, au sein desquels sont reçues des valeurs d'événements permettant de transiter vers (ou de rester sur) une implémentation pour chaque tâche reconfigurable. La combinaison des implémentations à un instant donné sera ici nommé *configuration*. La difficulté intervient lorsqu'il s'agit d'introduire des contraintes d'exécution nécessitant d'interdire la combinaison de certaines implémentations. Pour une raison quelconque, admettons que nous devions interdire la combinaison T2"' et T4" et revenons aux modèles comportementaux : si toutes les valeurs des événements des gardes proviennent de l'extérieur du système, alors manifestement le système seul est incapable d'assurer l'interdiction du couplage de T2"' et T4" (par exemple si les états activés sont

E2" et E4", rien n'empêche a priori la garde e23 d'être évaluée à *true* tout en conservant e41 à *false*, provoquant ainsi une transition vers la configuration interdite).

Dans un tel contexte, le concepteur a alors la possibilité d'introduire un nouveau modèle comportemental (figure 3), dont le rôle est d'influencer la valeur de certains événements des modèles comportementaux initiaux (par exemple l'un de ceux de la garde e23), événements que nous nommerons par la suite *contrôlables*, donnant ainsi du pouvoir au système dans le but d'assurer ses contraintes d'exécution.

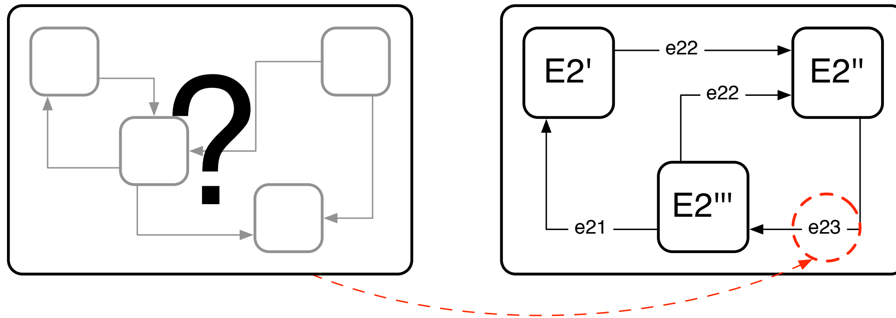


FIGURE 3 – La construction d'un modèle comportemental capable de valuer certains événements désignés afin de respecter des contraintes pour toute exécution est un problème difficile.

La conception sûre (respectant pour toute exécution les contraintes données par le concepteur) de ce nouveau modèle comportemental – qui sera nommé dans la suite par *resolver* – est un problème difficile qui sera détaillé dans ce document. Ce cas de figure où un système reconfigurable doit s'exécuter sous contraintes est typiquement ce que l'on rencontre dans le cadre des applications exploitant des architectures dynamiquement reconfigurables.

En effet, prenons l'exemple d'un FPGA rentrant dans cette catégorie : des zones reconfigurables du FPGA peuvent accueillir différentes implémentations (processeurs ou accélérateurs), implémentations exploitables par les tâches d'une application dont le modèle est alors assimilable au graphe reconfigurable que nous venons d'aborder (une tâche peut disposer d'implémentations logicielles ou matérielles, exploitant respectivement différents processeurs ou accélérateurs).

Le contexte particulier des applications exploitant les architectures reconfigurables implique la difficulté de conception que nous venons de relever lorsqu'il s'agit d'introduire des contraintes d'exécution impactant des combinaisons d'implémentations, ce qui justifie l'intérêt d'une approche à la fois sûre et intégrable dans un flot de compilation pour le déploiement de ce type d'application, intérêt constituant l'enjeu principal de cette thèse.

4 Plan

Ce document est organisé en trois parties, structurées de manière à présenter respectivement l'état de l'art, la contribution sous forme théorique, et l'exploitation de cette contribution sur un exemple concret.

La première partie donne donc un aperçu des notions d'architectures reconfigurables, d'outillages et de méthodes de conception sûre.

Le premier chapitre propose une vision d'ensemble des architectures visées pour ce travail en présentant tout d'abord l'aspect reconfiguration dynamique avec une vision haut niveau des principaux axes de recherches : principes de fonctionnement ainsi que leurs avantages et compromis vis-à-vis des autres architectures. Il expose ensuite les implémentations matérielles existantes des systèmes embarqués adaptatifs et notamment leur instanciation commerciale sous la forme de FPGA, avant de donner une vue d'ensemble des travaux exploitant la reconfiguration dynamique. L'objectif de ce chapitre est de présenter les enjeux de la conception d'applications utilisant ces architectures et d'en déterminer les caractéristiques communes afin de dresser le cahier des charges d'un dispositif de contrôle générique.

Le second chapitre présente ensuite les principaux axes de recherche concernant les architectures reconfigurables, et montre leurs principes de fonctionnement ainsi que leurs avantages et compromis vis-à-vis des autres architectures ; puis elle motive l'utilisation de la modélisation de haut niveau – en particulier l'ingénierie dirigée par les modèles (IDM), en comparant l'approche aux autres méthodes de spécification de systèmes embarqués. L'objectif est de montrer les avantages d'un profil UML, nommé MARTE, dans ce contexte : des travaux exploitant MARTE dans le cadre bien spécifique de la modélisation du contrôle de la reconfiguration dynamique pour des applications employant des architectures reconfigurables sont discutés, et des manques dans leurs flots de conception sont relevés ; manques qui serviront ensuite à justifier une approche de conception sûre.

Enfin, le troisième chapitre clôt l'état de l'art en présentant des techniques permettant d'augmenter la fiabilité des systèmes conçus, en s'intéressant notamment aux techniques formelles capables de garantir des propriétés pour toutes exécutions. Parmi ces techniques, l'une d'elle, la synthèse de contrôleur discret, est particulièrement mise en avant en raison de ses capacités à construire des solutions de contrôle d'après des contraintes. Un langage synchrone, BZR, est ensuite présenté et son choix justifié en raison de ses capacités à représenter des modèles comportementaux (tels que ceux introduits dans l'exemple informel précédent) et à servir de support à des outils de synthèse de contrôleur.

La seconde partie, constituant une présentation théorique de l'ensemble des contributions, est segmentée en trois chapitres (4, 5 et 6), qui traitent respectivement de la méthodologie de modélisation proposée, de sa transformation vers un modèle synchrone supportant la synthèse de contrôleur, et de ses possibilités d'optimisation en ligne par une exécution comprenant simultanément une partie purement réservée au contrôle de reconfiguration et une autre capable de choisir une configuration lorsque plusieurs sont légales.

Le quatrième chapitre se focalise donc sur une présentation des éléments de modélisation. L'approche proposée s'inscrit dans le cadre du langage MARTE, aussi un rappel de la sémantique des éléments MARTE exploités est effectué avant de poser la contribution à ce niveau, contribution se situant d'abord sur le plan de la méthodologie d'utilisation de MARTE pour la représentation du contrôle de reconfiguration, puis sur le plan de l'extension d'éléments existants afin notamment de définir ce qu'est un contrôleur et quelles sont les propriétés qu'il contrôle. Dans ce chapitre, seule la sémantique des éléments MARTE standards et étendus de la méthodologie est donnée, leur mise en pratique faisant l'objet de la troisième partie. À noter que le lecteur pourra s'interroger à ce niveau sur des aspects de type *représentation de l'architecture* (éléments physique ou virtuels) ou de type *modèle de communication* (échange d'information entre tâches, contrôleurs, etc.) : des abstractions de ces aspects sont faites dans la présente méthodologie, la raison étant qu'elle s'inscrit dans celle du projet FAMOUS, nommée RecoMARTE, dont les membres apportent chacun une contribution distincte dans le cadre d'un flot commun de modélisation/compilation. C'est pourquoi seuls les aspects de modélisation et de transformation des notions de contrôle de reconfiguration sont abordés dans ce travail.

La formalisation des transformations est d'ailleurs l'objet du chapitre cinq. L'automatisation du flot est une contribution majeure de ce travail : l'objectif est ici de donner les fondements formels permettant le passage d'un modèle en MARTE étendu respectant la méthodologie de modélisation vers un modèle synchrone, décrit en langage BZR, à partir duquel il est possible d'appliquer un outil de synthèse de contrôleur afin d'obtenir un code de contrôle correct par construction correspondant à la modélisation donnée par le concepteur. Le chapitre présente à ce titre un métamodèle dit *intermédiaire* faisant le lien entre un modèle RecoMARTE disposant d'informations de contrôle et une cible en BZR : des transformations sont ainsi réalisées de modèle vers modèle entre RecoMARTE et ce métamodèle intermédiaire assimilable à une représentation UML de BZR, puis de transformations directes depuis ce métamodèle vers BZR.

Enfin, le chapitre six présente la dernière contribution importante de ce travail : l'optimisation de l'exécution du contrôleur. Il s'agit de permettre au concepteur d'implémenter pour chaque contrôleur spécifié une fonction de décision dont le rôle est d'être informée

d'un espace de configurations accessibles – garanti par un contrôleur – afin d'en choisir une. Ce choix peut être purement arbitraire, ou bien relever d'une intelligence artificielle. La contribution repose donc ici sur la séparation des aspects purement contrôle, assurant formellement un comportement dynamique correct du système en restreignant ses possibilités d'évolution, et des aspects de décision, relevant quand à eux de la sélection d'une configuration dans un cadre pré-restreint. L'enjeu étant donc de montrer comment le mécanisme de contrôle formel peut communiquer avec un module dont le but est d'optimiser ses ordres de reconfiguration en travaillant sur un espace de possibilités d'évolutions.

La troisième et dernière partie est consacrée intégralement à la mise en pratique des contributions, validant ainsi l'approche par l'expérience. Elle est constituée de trois chapitres présentant respectivement une plateforme d'expérimentation matérielle et logicielle, un exemple de système modélisé conformément à la méthodologie proposée, et une démonstration de l'automatisation du flot de compilation.

Le chapitre six, débutant cette troisième partie, s'attache à définir une plateforme, à base de FPGA, capable de supporter à la fois la méthodologie proposée mais également l'ensemble des objectifs de FAMOUS : l'idée étant de définir une plateforme commune sur laquelle les contributions des différents membres du projet pourront aisément fusionner, avant de disposer en définitive d'une méthodologie complète capable de générer la plateforme elle-même. À ce titre, une plateforme reconfigurable capable de supporter la mise en œuvre d'applications dynamiques de traitement vidéo est proposée et justifiée.

Au septième chapitre, une modélisation d'un cas d'étude est donnée après avoir synthétisé les propriétés essentielles à présenter afin de démontrer l'ensemble de la méthodologie. L'enjeu est ici de proposer un exemple à la fois méthodologiquement complet et simple à suivre lors des traitements opérés tant dans sa modélisation que dans ses transformations.

Les transformations de cet exemple, ainsi que la simulation purement logicielle de son exécution sur la plateforme définie au chapitre six, sont mises en œuvre dans le neuvième et dernier chapitre, concluant ainsi la démonstration du flot de conception de cette étude.

En dernier lieu, une conclusion générale dresse un bilan des contributions et travaux accomplis avant de terminer le mémoire par une présentation des perspectives de ces travaux, tant en terme d'améliorations potentielles (questions ouvertes et débuts de pistes) que d'intégration dans le projet FAMOUS.

Première partie

État de l'art

1

Architectures reconfigurables

Sommaire

1.1 Introduction **25**

1.2 Principes et compromis **27**

1.2.1 Performance 27

1.2.2 Flexibilité 28

1.2.3 Classification 29

1.3 Systèmes embarqués adaptatifs **30**

1.3.1 Systèmes sur puce reconfigurables 30

1.3.2 Field Programmable Gate Array 32

1.3.3 Reconfiguration Dynamique et Partielle (RDP) 33

1.4 Études et travaux exploitant les architectures reconfigurables **34**

1.4.1 Prototypage 34

1.4.2 Mise à jour dynamique 35

1.4.3 Tolérance aux fautes 35

1.4.4 Contrôle thermique 36

1.4.5 Optimisation de la qualité de service 36

1.5 Conclusion **37**

1.1 Introduction

Traditionnellement dans le monde de l'informatique et de l'électronique, deux manières d'exécuter un calcul sont rencontrées : logicielle et matérielle. La manière matérielle, rendue possible par la conception d'un composant dédié tel qu'un Application-Specific Integrated Circuit (ASIC), permet d'optimiser les ressources afin de traiter rapidement des

tâches précises. Cette performance a cependant un prix : le coût de fabrication, souvent estimé en millions de dollars, allié à un effort important de conception (développement, test, débogage, etc.). La manière logicielle quant à elle, spécifiée sous la forme de programmes contenant des instructions interprétables par une unité de calcul générique (General Purpose Processor, GPP), dispose d'une flexibilité importante. En effet le support matériel générique facilite les changements d'applications et l'exécution d'un grand nombre de tâches différentes. À un prix également : celui de la performance, une tâche logicielle étant – et ce, de plusieurs ordres de magnitude – largement moins performante, en termes de rapidité d'exécution et de de consommation énergétique, comparé à son équivalent implémenté en matériel.

Les architectures reconfigurables (AR) appartiennent à un paradigme d'exécution à mi-chemin entre logiciel et matériel, combinant la flexibilité du premier à la haute performance du second. L'avantage peut être résumé simplement : la capacité de reconfiguration permet d'adapter le matériel après sa fabrication.

Le concept original des AR est attribué à Gerald Estrin [38], qui proposa en 1960 l'idée d'un ordinateur doté d'un processeur et d'unités matérielles reconfigurables. Mais cette idée n'eût pas immédiatement d'impact, faute de matériel adéquat pour la mise en œuvre du concept. Il faut attendre le début des années 1980 pour voir l'apparition progressive de matériel composé de logique programmable. Depuis, l'intérêt pour les AR n'a cessé de croître, notamment dans le domaine de la recherche, comme en témoignent l'augmentation rapide [63] du nombre de conférences dédiées à ce domaine et son adoption par des congrès tels que DAC, DATE, ISCAS, etc.

L'explosion des coûts de production d'un ASIC et, parallèlement, la réduction du cycle de vie des produits basés sur ce type de technologie ont en effet motivé la recherche pour tenter d'en remplacer au moins en partie par des architectures reconfigurables.

Mais la réduction des coûts, notamment ceux du débogage, du profilage de code, de la vérification, du tuning et de la maintenance, n'est qu'un facteur particulier de cet engouement. Le paradigme d'exécution offert par les AR a donné lieu à une révolution dans la manière de concevoir des systèmes afin d'exploiter au mieux cette notion de reconfiguration. De nouveaux langages, outils, techniques de compilations, etc. sont apparus, pour optimiser leur utilisation.

Ce chapitre s'attache à présenter les axes de recherche majeurs tirant parti des AR. La première section expose les principes et compromis des AR, d'abord en terme performance, puis en terme de flexibilité et classe les différents types d'AR définis dans la littérature. La deuxième section traite plus particulièrement des systèmes embarqués profitant de la reconfiguration. Enfin, la troisième partie aborde les études et travaux gravitant autour des AR.

1.2 Principes et compromis

Bien que les AR combinent les avantages du matériel en terme de performance, et du logiciel en terme de flexibilité, ces bénéfices sont cependant partiels, et sur le plan de l'efficacité d'exécution, l'utilisation des AR semble d'abord être une question de compromis performance/flexibilité.

1.2.1 Performance

Dans [11], K. Benkrid et al. effectuent une étude comparative des résultats d'implémentation d'un algorithme d'alignement de séquences d'ADN [1] sur diverses architectures. Il montre l'efficacité d'une implémentation dédiée au problème, réalisée grâce à une AR à base de Field Programmable Gate Array (FPGA), comparée aux autres implémentations exploitant des architectures plus génériques mais non-reconfigurables.

Ces autres architectures sont les suivantes : Un Graphics Processing Unit (GPU), permettant l'exécution efficace d'applications massivement multi-processus. Un Cell Broadband Engine (Cell BE) d'IBM, optimisé pour les calculs vectoriels à virgule flottante. Un General Purpose Processor (GPP), ici un processeur de la famille x86, l'architecture la plus répandue dans le monde des ordinateurs personnels, stations de travail et serveurs informatiques.

Les résultats présentent un net avantage de performance pour l'implémentation sur FPGA, celle-ci étant notamment de trois ordres de magnitudes plus efficace en terme d'énergie que celle sur GPP, pour une rapidité d'exécution plus de 200 fois supérieure.

Cependant, l'étude ne montre pas (pour une raison de coût) les performances d'une implémentation complètement matérielle. Dans [64], il est estimé qu'une même implémentation est généralement d'un ordre de magnitude plus rapide sur ASIC que sur FPGA, pour environ deux ordres de magnitudes d'efficacité en terme de surface.

Ainsi, si l'on considère uniquement la notion de performance pour un traitement donné, les AR occupent une place intermédiaire entre les architecture génériques, moins efficaces mais plus facilement exploitables de manière logicielle, et les architectures dédiées, très efficaces mais difficiles à concevoir et extrêmement coûteuses.

Cependant, jusqu'ici la notion de reconfiguration n'est clairement pas exploitée à son plein potentiel. En effet celle-ci n'est pour l'instant employée une fois pour toute qu'à la définition de l'application. Or, en prenant en compte la reconfiguration pendant le cycle de vie de l'application, il devient possible d'optimiser l'exécution par changement architectural. Ainsi, un nouveau critère démarquant les AR des autres architectures apparaît : la flexibilité.

1. algorithme de Smith-Waterman

1.2.2 Flexibilité

Dans un système reconfigurable, c'est-à-dire un système exploitant une AR, les parties matérielles et logicielles peuvent être modifiées suivant les spécifications du concepteur.

Du fait de cette nature, le concepteur de systèmes reconfigurables est amené à penser différemment des autres concepteurs. Typiquement, un concepteur de logiciels écrit des programmes séquentiels exploitant les capacités d'un micro-processeur à traiter des séries d'instructions. La conception d'un système reconfigurable requiert quand à elle de penser en terme de parallélisme spatial, c'est à dire en terme d'utilisation simultanée de ressources dispersées sur l'AR afin d'optimiser la charge de calcul.

Les concepteurs de matériel sont quant à eux déjà habitués à penser en ces termes, et sont donc avantagés pour adapter leurs méthodes de travail, mais la notion de flexibilité du matériel est pour eux une opportunité nouvelle. Elle représente un avantage à la fois statique et dynamique.

Statiquement parlant, la flexibilité matérielle des AR permet de prototyper rapidement un système, et même d'ajouter et changer des fonctionnalités au cours de la conception. Ce type d'avantage est généralement un luxe – coût de gravure, temps de développement – inaccessible pour le concepteur de matériel, qui consacre d'ailleurs en équipe un effort important dans la validation des besoins et la vérification des spécifications du système en amont afin de s'assurer qu'aucun changement ne sera nécessaire par la suite.

Dynamiquement parlant, la capacité d'un système reconfigurable à changer de configuration – et ce au cours de son cycle de vie – apporte quant à elle de nouvelles possibilités d'optimisations qui peuvent être :

- spatiales : une surface matérielle importante de l'AR peut être configurée à la demande pour effectuer un calcul donné ;
- ou temporelles : les ressources matérielles sont multiplexées dans le temps, permettant de créer de larges circuits sur une surface réduite.

Lorsque la notion de flexibilité est un besoin du système, et si ses différentes configurations ne peuvent être définies matériellement faute de place ou de coût, ou encore si celles-ci ne peuvent être connues à l'avance (complexité combinatoire, mises à jour et corrections de bugs, etc.), les AR sont alors une réponse adéquate face à des solutions dédiées non reconfigurables, qui par essence ne peuvent adresser ce besoin. Et si des questions de performances entrent en jeu à tel point qu'aucune solution à base d'architecture générique n'est envisageable, alors les AR et ne représentent plus un compromis. Elles deviennent même, dans ce cas précis, le seul choix possible.

1.2.3 Classification

Les AR ont en commun le fait d'augmenter les possibilités de répartitions spatiales et temporelles des calculs, au moyen d'un chemin de données reconfigurable, c'est-à-dire au moyen d'un ensemble modifiable d'unités fonctionnelles qui composent leur architecture.

Au-delà de cette caractéristique commune, plusieurs paramètres définis dans la littérature les distinguent. Chaque AR possède au moins l'une des propriétés suivantes :

La reconfiguration dynamique [86]

Cette caractéristique permet à une AR d'être modifiée à l'exécution, c'est-a-dire pendant que ses ressources sont utilisées. Pour qu'un tel concept soit possible, le processus de reconfiguration ne doit pas corrompre le calcul en cours, autrement dit il ne doit pas interférer avec le bon fonctionnement des ressources actives. Une architecture dynamiquement reconfigurable se distingue des autres AR par le fait que ces dernières doivent être configurées avant d'être activées (processus de reconfiguration "statique").

La reconfiguration partielle [77]

Une architecture partiellement reconfigurable dispose de la capacité à reconfigurer un ou plusieurs sous-ensemble(s) de ses ressources matérielles. Elle se distingue des AR à reconfiguration complète, dont la modification impacte nécessairement l'ensemble des ressources quel que soit le besoin d'adaptation. Cette notion de reconfiguration partielle est réalisable si l'interface de configuration de l'AR dispose d'un mécanisme pour mettre en place de nouvelles configurations ciblant précisément ces sous-ensembles de ressources.

La reconfiguration multi-contextes [36]

Dans un système dynamiquement reconfigurable, le temps de reconfiguration est généralement critique car celle-ci doit se produire à l'exécution. Une solution intéressante face à ce problème est d'embarquer des couches dans l'AR capables de stocker plusieurs configurations simultanément. Concrètement, une architecture reconfigurable multi-contextes dispose de plusieurs couches mémorisant des configurations, chaque couche pouvant être activée à différents moments. L'avantage des AR multi-contextes est de permettre des préchargements extrêmement rapides, de l'ordre de la nanoseconde [72], comparé à plusieurs millisecondes pour une reconfiguration d'AR de technologie équivalente mais à simple contexte. Les AR multi-contextes ont cependant l'inconvénient de nécessiter plus d'espace matériel que leur équivalent à simple contexte, espace normalement réservé pour la logique ou le routage. À titre d'exemple, une AR de type FPGA à 4 contextes ne pos-

sède que 80% de surface active pour la logique et le routage, comparé à son équivalent à simple-contexte [140].

La reconfiguration à grain fin ou épais

La granularité d'une AR, ou le grain de reconfiguration, désigne la taille de sa plus petite unité fonctionnelle reconfigurable. À grain fin, par exemple [142], une AR permet à un concepteur d'implémenter efficacement des tâches manipulants des données calibrées au bit près. Mais pour des traitements plus spécifiques et particulièrement intensifs et réguliers, les architectures à grains épais, par exemple PipeRench [45] ou RaPiD [35], sont généralement plus optimales (mais plus limitées quant à leur spectre d'application).

1.3 Systèmes embarqués adaptatifs

Les capacités des AR profitent à divers domaines, tel que le High Performance Reconfigurable Computing [37], la virtualisation matérielle [109] ou encore le prototypage de circuits [21]. Mais les systèmes embarqués, généralement soumis à des contraintes de performances pour un ensemble précis de tâches et de flexibilité pour s'adapter aux éventuels changement de leur environnement, constituent un domaine d'application particulièrement approprié pour les AR. C'est pourquoi la présente étude s'intéresse principalement à l'utilisation des AR dans un contexte embarqué.

1.3.1 Systèmes sur puce reconfigurables

Les systèmes sur puce, ou Systems-on-Chip (SoC), sont un paradigme de conception pour les système embarqués, émergeant depuis le début des années 2000 grâce à l'évolution des technologies à base de semi-conducteurs. Le principe du SoC est d'intégrer les composants essentiels d'un système sur une simple puce, cf. figure 1.1, tirée de Wikipédia [2]. Parmi ces composants essentiels intégrables, on retrouve généralement :

- des processeurs, de type GPP ou accélérateurs dédiés à un type précis de tâche ;
- des mémoires (RAM, ROM, Flash, etc.) ;
- des interfaces de communication (USB, Ethernet, PCI, etc.) ;
- des capteurs
- des convertisseurs numérique/analogique.

La principale différence entre un SoC classique et un SoC dit "reconfigurable" est la présence d'éléments reconfigurables parmi ses composants. Les SoCs classiques sont principalement conçus pour répondre à un besoin d'application donné, en respectant des

2. `http://en.wikipedia.org/wiki/System_on_a_chip`

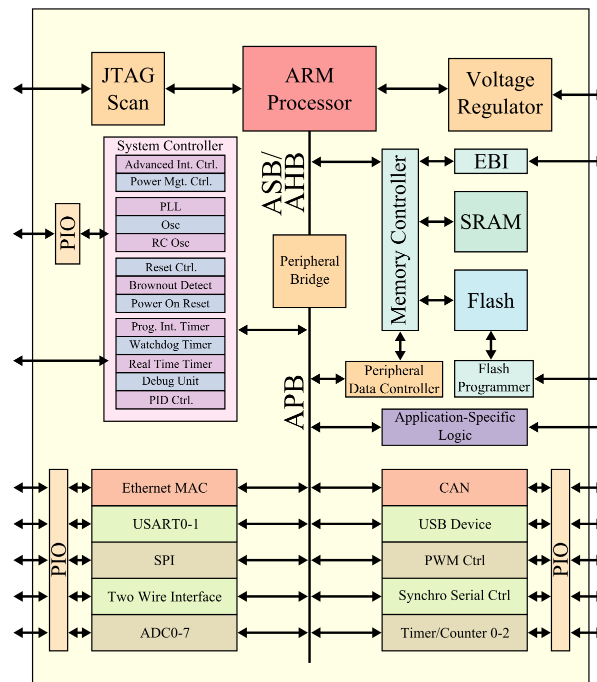


FIGURE 1.1 – Exemple de système sur puce, constitué d'un processeur ARM et de diverses interfaces de communication.

contraintes de performance élevées et souvent contradictoires (ex. vitesse d'exécution et économie d'énergie). Nous retrouvons les mêmes avantages et compromis soulignés dans la présentation des AR concernant les SoCs configurables lorsqu'ils sont comparés à leurs équivalents non reconfigurables : Un SoC classique est généralement coûteux à produire, dédié à un type d'application mais plus performant dans son domaine d'application comparé à un SoC équivalent reconfigurable, qui peut par contre profiter de ses capacités de reconfiguration pour s'adapter à différents cas d'applications. Les SoCs reconfigurables ciblent donc de préférence les systèmes où plusieurs applications doivent se partager les mêmes ressources matérielles, on parle alors de *virtualisation matérielle*. Littéralement, la virtualisation matérielle signifie qu'une application s'exécute grâce à un matériel d'exécution virtualisé, par opposition au matériel physique. "Matériel virtuel" semble être un terme contradictoire puisque la notion de matériel suppose une existence physique. De plus, la littérature emploie ce mot pour désigner plusieurs notions, notamment *l'exécution virtuelle*, les *machines virtuelles*, et le *p*artitionnement temporel [109].

La notion nous concernant ici est le partitionnement temporel : La motivation de ce concept réside dans le fait d'être capable d'exécuter une application d'une taille arbitraire sur un matériel de capacité insuffisante. Le principe étant de diviser l'application en parties suffisamment petites pour que chacune puisse s'exécuter sur ce matériel, puis d'exécuter ces parties séquentiellement. Le matériel doit donc être reconfiguré temporairement pour

chaque partie. Pour se faire, il doit disposer de blocs d'exécution programmables, ou *logique programmable*, et de connexions également programmables permettant aux blocs de communiquer leurs calculs entre eux. Ainsi, la partie physique du matériel est constituée de ces blocs et interconnexions, alors que la partie virtuelle désigne l'état de ces éléments. Éléments qui, une fois programmés, permettent d'exécuter une application.

1.3.2 Field Programmable Gate Array

La concrétisation des AR débute véritablement dans les années 1980 avec l'arrivée de matériels puissants dotés d'une logique programmable, appelés Field Programmable Gate Array (FPGA). De nos jours, les FPGA commerciaux, à grain fin et basés sur la technologie SRAM, dominent en tant qu'AR employées pour les SoCs reconfigurables.

Un FPGA est généralement constitué de deux couches, la première contenant la logique programmable, c'est-à-dire les blocs de calcul reconfigurables, associée à une hiérarchie de connexions également programmables permettant aux blocs de communiquer entre eux. Certains types de ressources étant jugés nécessaires pour un grand nombre d'applications, la plupart des FPGA intègre également dans cette couche un nombre variable de ressources supplémentaires, cette fois-ci non reconfigurables mais optimisées pour un besoin transversal particulier : blocs de RAM, processeurs de signal numérique ou DSP (Digital Signal Processor), multiplicateurs, etc. La seconde couche constitue la mémoire de configuration, dont le rôle est de contenir l'état des éléments reconfigurables de la première couche. L'écriture dans cette mémoire s'effectue au moyen de fichiers de configurations appelés *bitstreams*.

Traditionnellement lors de la conception d'un SoC reconfigurable à base de FPGA, la spécification de la partie matérielle à virtualiser s'effectue via un langage de description d'architectures, tel que VHDL ou Verilog. La partie logicielle quant à elle, composées de pilotes pour le matériel et/ou de programme(s) compilable(s) pour un ou plusieurs processeur(s), est généralement écrite en C, C++, voire en assembleur. Cette synergie de conception matérielle/logicielle se nomme *co-design* et est couramment menée via un outil de Conception Assistée par Ordinateur (CAO) pour l'électronique. Les constructeurs de FPGA conçoivent généralement ce type d'outils ciblant uniquement leurs produits (ex. EDK pour Xilinx, SoPC Builder pour Altera, etc.).

La phase suivant la conception matérielle/logicielle consiste, pour le matériel, à traduire les modules d'architecture spécifiés en descriptions de circuits appelées *netlists* puis de trouver leurs positionnements et interconnexions adéquats ("placement et routage" ou *place-and-route*) sur le FPGA ciblé. L'information de placement et de routage, en combinaison avec la description de circuit, est inscrite dans un fichier bitstream qui peut ensuite être téléchargé sur le FPGA afin de virtualiser le matériel correspondant.

Si la place mémoire (RAM) est suffisante sur le FPGA, le bitstream peut également contenir la partie logicielle sous forme de binaire compilé. Lorsque ce n'est pas le cas, la plateforme sur laquelle repose le SoC doit embarquer une telle mémoire (DDR, Flash, etc.), ou au moins disposer d'un accès à une mémoire adéquate, afin de stocker la partie logicielle.

Les architectures de type FPGA ont continuellement évolué depuis leur introduction, embarquant progressivement des ressources hétérogènes, différentes tailles de grain, etc. Et l'une des technologies supportée par certains FPGA modernes, la reconfiguration dynamique et partielle [15], a permis à de nombreux travaux (de recherche notamment) de se concrétiser matériellement. La présente étude se focalise essentiellement sur cette technologie, relativement jeune et assez peu exploitée à son plein potentiel.

1.3.3 Reconfiguration Dynamique et Partielle (RDP)

Dans le cadre d'un FPGA, la RDP désigne la modification en ligne d'une région reconfigurable spécifique de ce FPGA, sans impacter les autres régions qui peuvent continuer leur exécution. Les ressources physiques de cette région reconfigurable sont alors multiplexées temporellement afin de supporter plusieurs états mutuellement exclusifs, chacun permettant d'exécuter une ou plusieurs tâches logicielles.

Présentement, seuls Xilinx et Altera (deux fabricants majeurs de FPGA) fournissent des FPGA capables de reconfiguration dynamique et partielle. Dans les deux cas, la commande de reconfiguration peut être émise de manière interne, c'est-à-dire via un contrôleur embarqué dans le FPGA.

Xilinx est le premier à avoir mis en œuvre cette technologie, puis à avoir proposé une méthodologie d'utilisation, appelée *Early Access Partial Reconfiguration Flow* (flot EAPR) en 2006 [141].

Ce flot modulaire permet de diviser la conception en régions, chaque région ciblant une zone spécifique du FPGA cible ; une région peut donc être constituée de logique reconfigurable, de blocs de mémoire, de DSP, etc. Si une région est définie comme reconfigurable, elle devient une boîte noire, ou *black box*, ne montrant que son interface constituée de ports de communication. Pour une black box donnée, il est alors possible de définir plusieurs implémentations (circuits) concrètes, dont les contraintes sont 1) d'être implantables sur les ressources physiques réservées par la black box et 2) de respecter l'interface.

Une combinaison constituée d'une implémentation par région forme une configuration complète du FPGA. L'une de ces combinaisons doit être définie statiquement – formant un bitstream complet –, il s'agit de la configuration initiale, devant être téléchargée sur le FPGA à son démarrage. Depuis cette configuration initiale, il est alors possible – si la configuration initiale contient un contrôleur de reconfiguration – de déclencher des

reconfigurations partielles, c'est-à-dire de télécharger des implémentations (sous forme de bitstreams partiels) sur les régions correspondantes du FPGA.

À noter qu'il est important de maîtriser le moment de reconfiguration, afin de ne le déclencher que lorsque cela est techniquement et fonctionnellement correct. *Techniquement correct* signifie que la reconfiguration se produit seulement lorsque les régions impactées ont terminé leur traitement courant ou bien lorsqu'elles peuvent l'abandonner. Cet aspect étant généralement adressé via l'utilisation d'un modèle de calcul approprié [89], ou d'un mécanisme de gestion de tâches permettant leur interruption [138]. *Fonctionnellement correct* signifie que seules les reconfigurations vers des combinaisons d'implémentations autorisées peuvent se produire. Cet aspect nécessite de pouvoir définir sémantiquement la notion d'*autorisé*, afin d'employer des méthodes garantissant que rien d'interdit ne pourra se produire : tests exhaustif, preuve formelle, vérification formelle, etc.

1.4 Études et travaux exploitant les architectures reconfigurables

De nombreux projets (de recherche notamment) gravitant autour des AR ont vu le jour depuis l'essor des FPGA, particulièrement ceux disposant de la RDP. Cette section se focalise sur des domaines d'utilisation majeurs, tirant parti de cette capacité de reconfiguration offerte par les FPGA.

1.4.1 Prototypage

Le prototypage de circuits est le domaine d'utilisation industriel le plus ancien réservé aux FPGA [32, 131, 104], et toujours employé aujourd'hui [21, 5]. En effet, avant que les FPGA ne trouvent véritablement leur place en tant que matériel autonome pour répondre à un besoin de performance/flexibilité, leur capacité à virtualiser des circuits – et ce autant de fois que nécessaire après la fabrication physique du FPGA – est une véritable aubaine dans le monde de la conception matérielle.

Connaissant les coûts de conception extrêmement élevés d'un ASIC, il est obligatoire pour le concepteur d'effectuer un maximum de vérifications avant sa réalisation physique. Le prototypage d'un circuit sur FPGA, en vue d'être porté sur un ASIC, permet de le vérifier à moindre coût et au plus tôt (dès le commencement de sa conception) grâce au fait qu'il soit exécutable. Comparé à une approche de simulation purement logicielle, la réalisation sur FPGA permet d'obtenir des vitesses d'exécutions ainsi qu'une précision de modélisation au plus proche de la cible.

C'est pourquoi, aujourd'hui encore, les FPGA occupent également une place idéale en

tant que produit intermédiaire, permettant de diminuer les coûts de conception grâce à la simplification de la vérification.

1.4.2 Mise à jour dynamique

Le simple fait que la partie matérielle d'un système puisse être modifiée après sa conception physique rend les AR particulièrement intéressantes dans des cadres d'applications ou la mise en œuvre physique de l'ensemble des configurations nécessaires pour répondre à un potentiel besoin n'est pas envisageable, que ce soit pour une question de coût ou de taille des ressources. Dans ces cas de figures, il est alors envisagé de mettre à jour dynamiquement le matériel pour l'adapter au fur et à mesure aux besoins d'exécution.

Motivé par l'idée que ce type de cadre applicatif un jour répandu, le projet européen MORPHEUS (Multi-purpOse dynamically Reconfigurable Platformfor intensive HEterogenoUS processing) [133] a permis la mise au point [3] d'un prototype de processeur reconfigurable, embarquant plusieurs types d'AR. Pour l'instant la complexité d'utilisation est telle, comparée à une architecture classique à base de GPP, qu'une méthodologie de conception adaptée doit être proposée afin de rendre accessible son exploitation par un concepteur de systèmes reconfigurables.

D'autre cadres d'applications, beaucoup plus ciblés, rendent l'utilisation d'AR absolument nécessaire. Cela concerna notamment la mise à jour, voire la correction, de tâches matérielles embarquée sur des appareils à longue durée de vie et difficilement accessibles. [66, 68, 116] ont montré par exemple le besoin de pouvoir contrôler la reconfiguration matérielle dans le domaine des satellites et des missions spatiales longue durée.

1.4.3 Tolérance aux fautes

Il existe deux types de fautes concernant un système électronique : les fautes temporaires, provoquées par l'environnement sur le système, et les fautes permanentes, du fait de la dégradation (naturelle ou non) des composants dans le temps.

La tolérance matérielle aux fautes temporaires via des AR est notamment employée dans le contexte spatial, par exemple dans le cadre de l'exploration de Mars par des robots en 2004 [117]. En effet, lorsque le système de bord d'un Mars Exploration Rover détectait une panne logique [4] sur un de ses FPGAs [5], une reconfiguration complète du FPGA impacté était déclenchée, afin que le Rover puisse se réparer et reprendre sa mission.

---

3. par STMicroelectronics en 2009, `http://www.eetimes.com/electronics-products/processors/4111329/ST-rolls-Morpheus-reconfigurable-processor`

4. provoquée par un Single Event Upset, c'est-à-dire un changement d'état électronique dû à un rayonnement ionisant

5. FPGA Xilinx XQVR1000

Plus généralement, les rayonnements ionisants sont néfastes aux système électroniques [135]. C'est pourquoi l'emploi de FPGA dans un contexte spatial [13] – où ils peuvent être exposés au vent solaire, aux rayons cosmiques ou à la ceinture de Van Allen – a motivé la recherche en terme de tolérance aux fautes par le contrôle de la reconfiguration matérielle.

Les travaux présentés dans [108] traitent pour une partie de la gestion de la tolérance aux fautes mise en œuvre sur AR. Une architecture spécifique multiprocesseurs est notamment définie, chaque processeur pouvant se reconfigurer via un système de gestion de reconfiguration distribué dans lequel ils s'échangent des trames de vie afin de vérifier mutuellement leur bon fonctionnement. Si cette vérification échoue pour un processeur, une phase de reconfiguration est initiée afin de tenter de le réparer, puis de reprendre son exécution. Si la faute est permanente, un mécanisme de migration des tâches qui lui étaient allouées intervient pour transférer la reprise d'exécution vers un processeur sain.

1.4.4 Contrôle thermique

La dissipation de chaleur pendant l'exécution est également un problème rencontré dans les systèmes embarqués. Et si des méthodes de ralentissement d'exécution telles que l'ajustement dynamique de la fréquence et de la tension (Dynamic Voltage and Frequency Scaling) [110] sont applicables pour des architectures non reconfigurables, les FPGA dynamiquement reconfigurables apportent un nouvel axe de réponse à ce problème :

Le multiplexage temporel, classiquement réservé à l'optimisation des besoins en terme de ressources matérielles, a également été employé avec succès dans [79] afin de prendre en compte la notion de dissipation thermique d'un tâche lors de son exécution. Les capteurs présents dans le système embarqué donnent une information de répartition de la chaleur et permettent au système de contrôler thermiquement le FPGA par reconfiguration matérielle vers des tâches moins génératrices de chaleur.

1.4.5 Optimisation de la qualité de service

Si l'emploi de la reconfiguration matérielle permet d'améliorer la durabilité du FPGA dans un contexte hostile (chaleur, irradiation), elle peut également être employée pour augmenter les performances d'exécution. Il s'agit ici de la notion générique de *qualité de service*. Dans [31], Diguet et al. définissent notamment trois critères essentiels de qualité de service :

- La consommation d'énergie
- La rapidité d'exécution
- La quantité de ressources physiques nécessaires (surface, mémoire, etc.)

En disposant d'information quantitative sur ces critères pour chaque implémentation des tâches élémentaires d'un système reconfigurable donné – chaque tâche dispose de deux implémentations possibles : matérielle ou logicielle –, Diguet et al. ont ensuite proposé une méthode algorithmique permettant de reconfigurer les tâches du système afin qu'il s'adapte en ligne à des consignes de qualité de services arbitraires. Bien que l'utilisation des FPGA ne soit pas adressée – la plateforme étant composée d'un GPP et d'accélérateurs matériels –, des liens directs peuvent être aisément établis entre la l'adaptation en ligne d'une tâche et le contrôle de la reconfiguration dynamique et partielle.

À noter que les travaux dans [31] ont également apporté une contribution dans le domaine des services de systèmes d'exploitation – Operating Systems (OS) – pour la gestion de la reconfiguration. Il s'agit d'un domaine porteur pour l'exploitation poussée des AR, permettant la simplification de l'accès à la technologie (abstraction vis-à-vis du mode d'exécution – logiciel ou matériel – des tâches du système, virtualisation des communications afin d'abstraire la localisation des tâches, etc.).

À ce titre, nous pouvons également mentionner le projet FOSFOR [101], qui avait notamment pour objet l'étude de services d'OS [30, 103] (ordonnancement, communications, mémoire) capables d'augmenter la flexibilité et la scalabilité d'un système reconfigurable – dans le but d'assurer une qualité de fonctionnement – en faisant de la reconfiguration dynamique un aspect central.

Dans ces deux travaux mentionnés, la notion de qualité est l'objectif à atteindre : elle devient une donnée mesurable à partir de laquelle les services d'OS s'adaptent afin d'assurer une qualité de service, au moyen – si besoin – de la reconfiguration dynamique. Une telle approche permet au concepteur de se concentrer essentiellement sur les objectifs de son système, en l'assistant par la prise en charge d'hypothèses relevant de l'exploitation des propriétés d'AR (adaptation dynamique pour le respect de la qualité, etc.).

1.5 Conclusion

Si l'utilisation des FPGA en tant qu'outil pour prototyper un système non reconfigurable a mis en valeur la notion de vérification, afin d'assurer la fiabilité du circuit conçu, les autres travaux évoqués à la fin de ce chapitre – exploitant la reconfiguration dynamique et partielle – ont en commun la notion de contrôle de la reconfiguration.

Génériquement, ils peuvent tous être définis dans le cadre d'exécution présenté en figure 1.2 : l'exécution de chaque système considéré est assimilable à une boucle d'exécution infinie, dans laquelle des capteurs obtiennent des informations de la part de l'environnement du système, puis les communiquent à un module – le contrôleur de reconfiguration – dont le rôle est de sélectionner une configuration adaptée pour le système. Si la demande

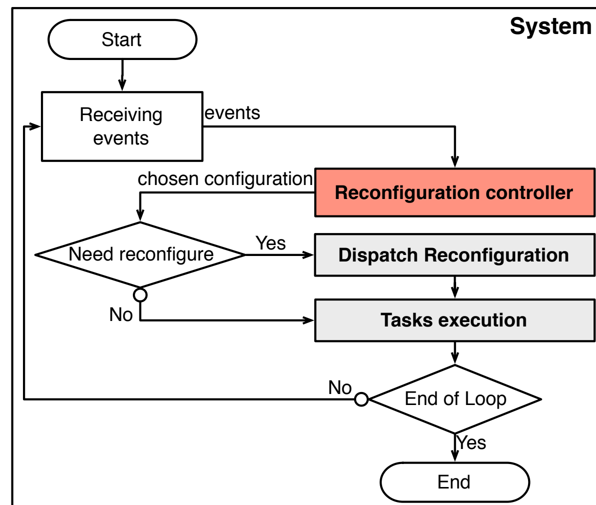


FIGURE 1.2 – Principe d'exécution générique avec contrôleur de reconfiguration

de configuration donnée par ce module est différente de la configuration courante, alors il se produit un chargement d'un ou plusieurs bitstreams approprié(s) sur le FPGA. Après reconfiguration, ou s'il n'y avait pas besoin de reconfigurer, le système peut exécuter une séquence de tâches, puis demander à ses capteurs d'obtenir de nouvelles informations de l'environnement. Cette figure fait notamment la synthèse du mécanisme de contrôle et de propagation générique d'un ordre de reconfiguration dans un système dynamiquement reconfigurable présenté dans [143].

De par leur capacité à s'adapter rapidement aux changements de contextes, les AR – en particulier les FPGA – sont particulièrement adaptées dans les domaines évoqués en section 1.4. Cependant, et nous l'aborderons plus en détails dans les prochains chapitres, le manque de méthodologie à la fois unifiée et efficace pour leur exploitation rend leur usage difficile. De plus, la fiabilité du circuit devant être assurée y compris dans un contexte dynamique, la sûreté du contrôle de reconfiguration est un besoin transversal dans tous les travaux exploitant la reconfiguration dynamique, ce qui augmente encore la complexité de conception.

Dans la suite, nous serons motivés par deux éléments clés permettant de simplifier et sécuriser la conception de ses systèmes : 1) les outils et moyens de conception employés pour les exploiter en tirant parti de leurs capacités d'adaptation, et 2) les méthodes appliquées pour garantir les aspects de sécurité dans le contrôle de reconfiguration.

2

Co-conception de systèmes sur puces

Sommaire

2.1 Introduction **39**
2.2 Modèles, méthodes et outils **41**
2.2.1 High Level Synthesis 41
2.2.2 Domain Specific Languages 42
2.2.3 Model Driven Engineering 43
2.3 MDE pour la conception de SoC **49**
2.3.1 Profils UML 49
2.3.2 UML/MARTE en particulier 50
2.3.3 UML/MARTE pour la modélisation des SoC reconfigurables 53
2.4 Conclusion **56**

2.1 Introduction

La conception d'un système sur puce désigne le développement en parallèle de ses aspects logiciels et matériels. Le besoin en terme d'abstraction n'a jamais cessé d'augmenter dans ce domaine, particulièrement depuis l'arrivée des circuits intégrés en 1950. L'avènement du *Computer Aided Engineering* à la fin des années 1960, avec de rudimentaires simulateurs de logique, a été une première réponse à ce besoin. Cela permettait typiquement de tester des circuits à partir de netlists (représentations textuelles de circuits), en générant des stimulis afin d'obtenir des résultats de simulation.

Puis, les circuits intégrés devenant de plus en plus complexes avec l'augmentation progressive du nombre de leurs portes logiques, les temps de simulation ont explosé de manière exponentielle. C'est alors que les premiers éditeurs de circuits font leur apparition

au début des années 1970. Par leur approche visuelle et numérisable, les représentations de circuits proposées par ces éditeurs permettent de simplifier la conception [6] et la vérification [7], ce qui contribue à diminuer les besoins de simulation [18]. On considère ces éditeurs comme les ancêtres des outils de Conception Assistée par Ordinateur (CAO).

C'est en 1980 que la motivation d'employer des langages afin de décrire des architectures apparaît [96], on parle de *Hardware Description Language* (HDL). L'idée est de spécifier une architecture sous forme d'un programme compilable vers une représentation de circuit. L'objectif est de capter l'expressivité du monde du logiciel vers le matériel : à la manière des langages pour le logiciel, il s'agit de gagner en niveau d'abstraction. Cela permet ainsi un accroissement en complexité des systèmes pouvant être conçu en un temps raisonnable. Deux HDL ont particulièrement émergé, il s'agit de Verilog, introduit par Gateway en 1986, et de VHDL, défini en 1987 par le département de défense des États-Unis. À la suite de leur introduction, des simulateurs de HDL, tels que ModelSim [46] et Cadence [69], ont rapidement été développés, permettant de tester et vérifier des circuits simplement en se basant simplement sur leurs spécifications en HDL.

Par la suite, trois principales approches modernes d'abstraction se sont développées. La première, la Synthèse de Haut niveau, s'inscrit dans la continuité des méthodologies d'abstractions : elle consiste à cibler une description d'architecture (niveau d'abstraction courant offert par les HDL) à partir d'un langage de programmation permettant de masquer de nombreux aspects d'implémentation bas-niveau. On parle d'approche *bottom-up*, dans le sens où l'on part de ce qui est connu pour en réaliser une abstraction directe. La seconde, l'approche par Domain Specific Languages (DSL), a pour objectif de définir un langage de programmation *idéal* afin d'adresser un problème particulier de conception. Enfin la troisième, l'Ingénierie Dirigée par les Modèles (IDM), a pour mission de définir le plus haut niveau possible d'abstraction, permettant la conceptualisation pure, et souvent visuelle, d'un système sans se préoccuper des détails d'implémentations. Ces derniers étant ensuite intégrés au fur et à mesure de la conception au moyen de raffinements appelés *transformations de modèles.*

Ce chapitre s'attache à détailler ces trois approches. De par sa capacité à faire communiquer voire fusionner le monde de l'analyse des systèmes et celui du développement, l'accent est particulièrement mis sur l'Ingénierie Dirigée par les Modèles.

---

6. le placement et routage de composants est nettement plus simple car physiquement proche de l'objectif

7. par l'emploi d'outils formels capables de déterminer si un circuit satisfait un ensemble de contraintes appelées *règles de conception*

2.2 Modèles, méthodes et outils

2.2.1 High Level Synthesis

La synthèse de haut niveau, ou *High-Level Synthesis* (HLS) [24], est un processus de conception pour la génération de systèmes numériques à partir de représentations comportementales. Ces représentations sont par exemple spécifiées en tant que sources en C/C++ ou SystemC [132]. Le rôle d'un outil de HLS est de compiler ces sources vers une représentation intermédiaire, nommée *Register Transfer Level* (RTL), capable d'être implémentée par des outils de synthèse logique traditionnels (ces derniers transforment le code RTL en *netlist*, c'est-à-dire en description de circuit imprimé, physiquement plaçable et routable sur une plateforme cible).

Usuellement, un code RTL décrit le flot de signaux, et son contrôle, entre les registres et la logique combinatoire. Ces descriptions de niveau RTL donnent une représentation comportementale temporisée, c'est-à-dire qu'à chaque pas de contrôle, il est nécessaire de définir les opérations – issues de la représentation comportementale – devant être exécutées.

Comparée au code RTL, une représentation comportementale de haut niveau s'affranchit généralement de la temporisation (bien qu'il soit possible dans certains cas, par exemple en SystemC, d'utiliser des instructions liées au temps). Elle ne contient donc pas de notion de temps, et seul l'ordre des opérations est fixé. En se basant sur des contraintes imposées par le concepteur, l'outil de HLS a pour mission de compléter (de "synthétiser") ces détails manquants, en déterminant par exemple à quel cycle doit s'exécuter une opération, dans quelle unité de mémoire doit être stocké un résultat, comment réaliser le contrôle du flot de signaux, etc.

Les outils de HLS opèrent généralement sur un large espace de conception, dans lequel la meilleure solution d'implémentation finale est recherchée. L'espace de conception est constitué des différents types de ressources sur lesquelles les opération sont exécutées, de l'espace disponible sur la puce pour instancier ces ressources, et de l'ordre dans lequel les opérations peuvent être exécutées. Différentes solutions acceptables dans un tel espace peuvent résulter en différentes implémentations avec différentes structures et séquences de contrôle.

L'emploi de la HLS est très avantageux dans le cadre de la conception de SoC reconfigurables, dont les espaces de conception, prenant en compte les reconfigurations, sont susceptibles d'être larges, rendant l'approche bas niveau (spécification comportementale temporisée) difficilement appréhendable à la main. En effet, dans l'approche bas niveau, les optimisations réalisables par un outil de synthèse logique concernent simplement l'utilisation des ressources et la latence du circuit, mais pas les coûts de reconfiguration.

Le concepteur est chargé d'optimiser manuellement ses besoins de reconfiguration. L'approche haut niveau via HLS offre quant à elle l'avantage de rechercher automatiquement une solution dans l'espace de conception à partir de contraintes et compromis entre l'utilisation des ressources, la latence d'exécution, et également les coûts de reconfiguration.

Les travaux autour de la synthèse de haut niveau pour le reconfigurable, motivés par l'élévation du niveau d'abstraction lors de la conception, sont nombreux et les spécifications comportementales de haut niveau servent autant à la conception de systèmes [25], que comme modèles pour l'analyse et la vérification [26]. L'avantage logique de la HLS est celui dû à la notion d'approche bottom-up, qui confère à la HLS un haut degré d'intégration vis-a-vis des méthodologies et outils existant [130]. Mais dans les faits, le niveau d'abstraction offert n'est généralement pas suffisant pour être pleinement indépendant des détails d'implémentation bas niveau. Plusieurs inconvénients résultent de l'emploi de SystemC ou d'un langage similaire afin de spécifier le système (architecture et application). Les informations du système relevant notamment de la hiérarchie structurelle, du parallélisme et des dépendances de données sont difficilement représentables ; et l'usage d'une représentation textuelle rend également complexe la différentiation des concepts du système, impactant du même coup négativement les temps de maintenance, d'évolutivité, et finalement de time-to-market [111].

2.2.2 Domain Specific Languages

Adresser fondamentalement l'ensemble des difficultés liées à la conception de SoCs grâce à une méthodologie, un outil, un langage, etc. est une problématique complexe. C'est pourquoi l'approche consistant à se focaliser sur un aspect précis de la conception en faisant abstraction des autres permet d'adresser partiellement mais efficacement la gestion de cet aspect, participant ainsi à la simplification du système dans son ensemble.

L'approche spécifique consistant à développer un langage dédié à un domaine d'application, ou *Domaine Specific Language* (DSL) [98], est un classique pour adresser la complexité du domaine ciblé, comparé à l'emploi d'un langage général (*General-purpose Programming Language* (GPL) [12] sans librairie particulière, par exemple C/C++, Java, Python, etc.. (Cela dit, combiné à une librairie dédiée, tout GPL peut se comporter tel un DSL ; l'Application Programming Interface (API) [8] de la librairie constituant alors le vocabulaire spécifique au domaine ciblé.)

En fournissant des notations et moyens de spécification dédiés à un domaine précis, un DSL échange cette notion de généralité propre aux GPL afin de gagner en expressivité sur ce domaine.

8. Une API est une interface fournie par un composant logiciel afin d'interagir avec lui.

Concernant le domaine de la conception de SoC en particulier, on relève notamment des approches facilitant l'analyse et la vérification en forçant le concepteur à spécifier son système dans un modèle formel, par exemple sous forme de machines à états finis comme c'est le cas dans [1]. D'autres approches se focalisent plutôt sur l'exploitation du parallélisme inhérent au matériel en permettant d'exprimer les systèmes dans un modèle de calcul adapté ; à titre d'exemple, Array-OL [43] est un DSL multidimensionnel permettant de décrire le parallélisme potentiel dans un système ; il est particulièrement approprié dans le cadre de la spécification de traitements intensifs du signal, et c'est en même temps un modèle de calcul disposant d'une sémantique pour la répartition en parallèle de traitements opérant sur des matrices de données.

Le cadre d'utilisation d'un DSL étant par essence restreint, l'emploi d'un DSL pour la spécification d'un système adresse de manière incomplète sa complexité, puisque le gain d'expressivité s'effectue au détriment de la généralité. Ce type d'approche n'est donc pas autonome et doit s'interfacer avec le reste du flot de conception afin de produire un système complet.

2.2.3 Model Driven Engineering

L'ingénierie dirigée par les modèles, ou *Model Driven Engineering* (MDE), est un concept issu du domaine du logiciel [75]. Les systèmes logiciels s'avérant de plus en plus complexes à mettre en œuvre, l'automatisation des processus de développement est devenu une opération primordiale afin de minimiser les coûts de conception et de maintenance tout en améliorant le temps de mise sur le marché.

Parmi les diverses approches dédiées à cette automatisation, la modélisation, notion sous-jacente du MDE, semble être la plus prometteuse [126]. Elle est à cet effet employée pour à la fois représenter les aspects du système de manière simple en faisant abstraction d'un maximum de détails d'implémentation, et mécaniser le processus de conception à partir de ces représentations appelées *modèles*.

De par son efficacité démontrée pour le logiciel, l'emploi du MDE tend à se développer dans le domaine de la conception logicielle/matérielle, ainsi que dans la conception matérielle en général, en raison des similarités en terme de difficultés de conception.

Le MDE se démarque des approches bottom-up telles que la synthèse de haut niveau par le fait que l'on s'intéresse d'abord à conceptualisation rapide du système, avant de le diviser en sous-parties aisément manipulables. Il s'agit donc à l'inverse d'une approche analytique dite *top-down* où, en partant d'une vision d'ensemble basée sur des représentations simplifiées des différents concepts du système, on décompose celui-ci en éléments toujours plus détaillés pour déboucher sur une implémentation finale.

Par son aspect holistique, le MDE se démarque également des approches ciblées, de

type DSL. En effet, il s'agit de diriger intégralement la conception du système depuis un ensemble de modèles capables d'adresser ses différents aspects. À noter que la frontière est souvent floue entre DSL et MDE, et pour cause, ses deux notions sont souvent combinées afin bénéficier du meilleur des deux mondes : un DSL dispose en effet d'un métamodèle qui le décrit, celui-ci pouvant ce titre être dérivé en tant que modèle d'un autre métamodèle plus abstrait, ce qui correspond à une approche MDE pour la conception de DSL. Un exemple de ce type d'approche est donné dans [76], où un métamodèle de syntaxe abstraite de DSL est défini pour permettre la spécification de DSL concrets dont les syntaxes et domaines peuvent être très différents, tout en partageant certains aspects tels que le fait d'être des langages impératifs, ou de disposer d'un flot de contrôle. Cette combinaison MDE/DSL permet notamment aux auteurs de réutiliser de nombreux composants communs à la définition de multiples DSL.

Le MDE a pour objectif de simplifier un flot de conception en s'intégrant à celui-ci au niveau de l'analyse, c'est-à-dire dès le commencement de la conception. Il ne remplace donc pas les méthodes de plus bas-niveau (typiquement, la synthèse de haut niveau est une méthode bas-niveau comparée au MDE) mais tend à simplifier au maximum leur usage [82] en permettant au concepteur de ne les utiliser que lors de la réalisation des niveaux de détails les plus fins (exemple : implémentation d'une fonction), lui retirant ainsi le besoin d'implémenter les aspects haut-niveau (tels que la structure du système, les communications inter-tâches, les mappages de tâches sur des processeurs, etc.) ceux-ci étant générés par raffinement, ou *transformation de modèle.*

Modèles

Les modèles sont la notion clé du MDE, dans laquelle des *concepts* représentant des réalités du système (c'est-à-dire des éléments distincts, dont on peut donner une interprétation) sont définis et liés par des *relations*. Un modèle est une représentation abstraite d'un aspect du système modélisé. Elle est abstraite dans le sens où les détails non nécessaires du système – concernant l'aspect modélisé – sont omis pour plus de clarté.

La notion d'abstraction existe depuis le début de l'histoire du logiciel, un logiciel étant par essence une abstraction de manipulations d'éléments physiques dans un machine. Le code machine, c'est-dire l'emploi de nombres binaires interprétables par une machine, a ainsi été la première abstraction réalisée dans ce sens. Toutefois, l'utilisation de nombres binaires devenant vite fastidieux pour le concepteur, les langages assembleurs ont été créés pour les remplacer par des instructions. Un code assembleur, abstraction d'un code en binaire, étant à la fois facilement transposable en binaire et plus compréhensible d'un point de vue humain. Bien que les langages assembleurs aient simplifié la tâche du concepteur, leur apprentissage nécessitaient de connaitre les instructions précises reconnues par le ma-

tériel utilisé afin de créer des programmes corrects. L'indépendance vis-à-vis du matériel est alors devenu une motivation majeure pour simplifier la conception. Les langages dits de haut niveau tels que C, Lisp, Pascal, ou Fortran ont été créés dans ce sens. Un même programme décrit dans l'un de ces langages peut être transformé en un code machine interprétable par le matériel ciblé au moyen d'un compilateur. La compilation étant la transformation d'un code abstrait vis-a-vis du matériel en un code concret et exécutable pour un matériel donné.

À ce stade de l'évolution du logiciel, les problèmes de conception viennent maintenant des coûts de développement, de maintenance, d'évolution, etc. L'analyse et la conception orientée objet, Object-Oriented Analysis and Design (OOAD) [125] est alors apparue dans le but de palier à ces coûts. Dans l'OOAD, les objets sont des entités indépendantes interagissant au sein du système. Ils disposent chacun d'un état et fournissent des opérations. Ces objets sont définis par des *classes* qui spécifient leurs attributs – c'est-à-dire le domaine de valeurs d'états –, et leurs opérations. Les classes permettent également de définir des notions d'héritages d'attributs et d'opérations entre elles, d'encapsulations de classes ou hiérarchies, et de polymorphisme soit la spécification abstraite d'opérations définissables par des classes qui en héritent. Les éléments du système définis par ces concepts ont alors la capacité d'être hautement modulaires et réutilisables.

L'approche OOAD suppose la définition de classes d'objets et de relations entre ses classes. Ces notions de classes et d'objets impliquent la séparation entre la spécification et l'implémentation de la spécification : à l'exécution, les objets sont créés dynamiquement d'après la définition donnée par leur classe et ses relations. Classes, objets et relations sont des abstractions structurelles d'un système donné, implémentables via des langages objets, par exemple Smalltalk, Ruby, Java ou Python, et interprétables par une *machine virtuelle* appropriée. Contrairement au processus de compilation qui fournit un code exécutable pour une machine donnée, une machine virtuelle permet d'exécuter un même code sur tout matériel [9] en gérant dynamiquement des détails d'exécution bas niveau, notamment la gestion de la mémoire physique dédiée aux objets.

À la différence des classes et objets de l'OOAD, les modèles ne requièrent pas de prendre en compte des aspects particuliers d'implémentation, et sont ainsi encore plus flexibles et abstraits. Ils sont le lien entre le domaine des langages de programmation (particulièrement les langages objets), et celui de l'analyse et la conceptualisation des systèmes. Le MDE offre à cet effet un cadre de conception dans lequel des modèles tout à fait adaptés au raisonnement analytique sont définis en amont de la conception, puis transitent jusqu'à un état productif où ils deviennent les éléments concrets d'un système, grâce à une sémantique qui leur est associée.

9. Tout matériel disposant d'une implémentation de la machine virtuelle

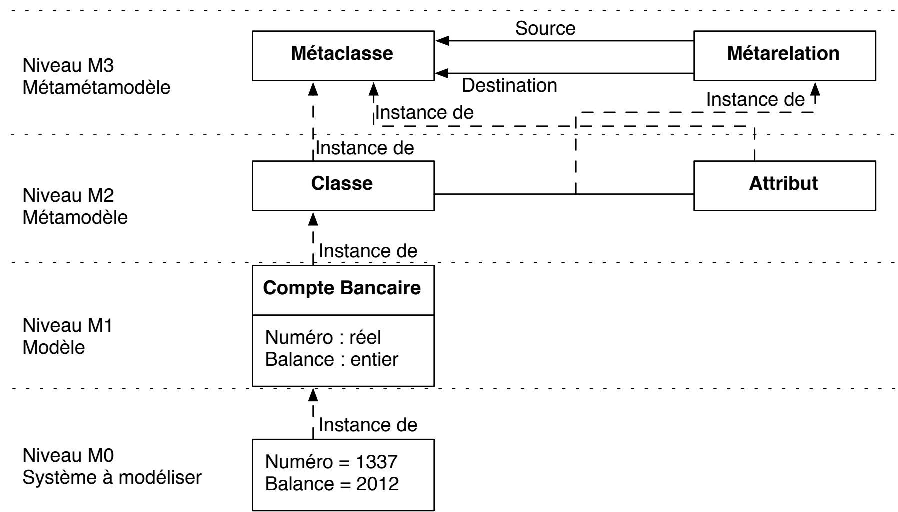


FIGURE 2.1 – Niveaux de modélisation du MDE et exemples de modèles

La sémantique d'un modèle donné est définie dans un *métamodèle* – littéralement un modèle du modèle – auquel il se conforme.

Métamodélisation

Afin de favoriser l'automatisation de leur transition jusqu'à un état concret dans le système ciblé, les modèles doivent disposer d'une sémantique formelle, fournie par leur métamodèle. Dans l'approche MDE, un métamodèle est la définition de l'ensemble des concepts et des relations – autrement dit la syntaxe – permettant de concevoir un modèle.

Un modèle conçu d'après la syntaxe spécifiée par un métamodèle est dit *conforme* à ce métamodèle de niveau d'abstraction supérieur. Comme son nom l'indique, un métamodèle est lui-même un modèle. Il se conforme donc à son tour à un métamodèle de niveau d'abstraction supérieur.

Le moyen formel de terminer cette récursivité particulière est de définir un métamodèle conforme à lui-même, c'est à dire capable de se définir grâce aux seuls concepts et relations qu'il définit. Un tel métamodèle est appelé métamétamodèle. *Unified Modeling Language* (UML) [50] et *Meta Object Facility* (MOF) [47] sont des exemples largement utilisés de métamodèle et métamétamodèle.

La figure 2.1 présente les quatre niveaux rencontrés dans la démarche de modélisation. Au niveau M0, nous retrouvons une représentation concrète, réelle, des objets d'un système. À titre d'exemple, nous voyons plusieurs variables, *Numéro* et *Balance*, auxquelles sont attribuées des valeurs. Au niveau M1, premier niveau d'abstraction, nous retrouvons les concepts spécifiés par les développeurs du système, autrement dit le modèle du système. Ici le concept de *Compte Bancaire* est défini, déclarant les variables *Numéro* et

Balance, ainsi que leur type (nombre réel et entier), utilisables au niveau zéro. Ce modèle de niveau M1 est conforme à un métamodèle que nous retrouvons au niveau M2. Celui-ci définit la notion de conteneur générique, nommé *classe*, capable d'encapsuler des déclarations de variables appelées *attributs*. Le compte en banque est ainsi une instance particulière de classe ayant deux attributs : *Numéro* et *Balance*. Enfin, le métamodèle de niveau M2 se conforme à un métamétamodèle décrit au niveau M3. Ce métamétamodèle expose les notions fondamentales de concepts et de relations, respectivement nommées *métaclasse* et *métarelation*. Les notions de classe et attribut issus du niveau M2 étant des concepts du systèmes, ce sont alors des instances de métaclasse, et le lien de dépendance établis entre classe et attribut étant une relation, il s'agit donc d'une instance de métarelation.

Le métamétamodèle de niveau M3 est défini de manière à se conformer à lui-même. *Métaclasse* et *métarelation* étant des concepts de ce modèle, ce sont à leur tour des instances de métaclasse. De même pour les liens de dépendance *source* et *destination*, instances de métarelation.

Lorsque qu'un métamodèle se conforme à lui-même (c'est donc un métamétamodèle), et que sa syntaxe peut être décrite formellement, alors les modèles se conformant récursivement – c'est-à-dire directement ou indirectement – à ce métamodèle peuvent être interprétés par une machine, l'objectif étant de les transformer automatiquement en des représentations plus concrètes, prenant ainsi du sens au sein même du flot de développement.

Transformations de modèles

Tout l'intérêt du MDE réside dans la notion de transformation de modèles. À l'instar du domaine de l'analyse des systèmes, le MDE est une aide à la compréhension et à la communication grâce aux modèles. Mais l'avantage supplémentaire du MDE vient de son utilisation dans le cadre de la génération de résultats concrets pour le développement, tels que des codes sources.

Le principe est qu'un ensemble de modèles, dont la sémantique formelle est reconnue par une machine, peut être traité en tant que source de règles de transformations afin de générer un ou plusieurs modèle(s) cible(s) se conformant à son (ou leur) tour à un ou plusieurs métamodèle(s). Les cibles peuvent donc être des modèles intermédiaires, à partir desquels d'autres règles de transformations peuvent s'appliquer, et ainsi de suite jusqu'à obtenir un modèle final ayant du sens dans le flot de développement, typiquement un code source.

De telles transformations sont généralement unidirectionnelles, dans le sens où seul l'ensemble de modèles source peut être modifié par l'utilisateur, les modèles cibles étant

re-générés en conséquence. Dès lors qu'un modèle cible est modifié, la synchronisation avec l'ensemble des modèles sources est perdue car une nouvelle transformation écraserait les modifications apportées. Des techniques existent afin de séparer formellement les parties générées d'un modèle cible de celles réservées à la modification [10]. Ainsi en cas de modification de la cible, seules les parties générées sont re-synchronisées après une nouvelle transformation, ce qui ne détruit donc pas les éventuelles parties modifiées. On parle alors de *transformations bidirectionnelles*, dont le principal challenge est d'assurer la synchronisation entre modèles sources et cibles. [27] propose un aperçu très complet des travaux sur les transformations bidirectionnelles, montrant leur emploi dans des domaines variés tels que les interfaces graphiques, les bases de données relationnelles, mais également et surtout : l'Ingénierie Dirigée par les Modèles. [146] formalise l'invariance de la traçabilité, c'est-à-dire le maintien du lien de synchronisation après modification de sources et cibles, au moyen de transformations bidirectionnelles faisant usage de balises dans les modèles générés.

Model Driven Architecture

L'Architecture Dirigée par les Modèles, ou *Model Driven Architecture* (MDA), est la démarche standardisée de MDE la plus connue [100]. Elle est proposée par l'Object Mangement Group (OMG), qui en détient la marque [48].

L'approche MDA regroupe de nombreux standards, dont Meta-Object Facility (MOF) qui est un métamétamodèle, le métamodèle Unified Modeling Language (UML), conforme à MOF et dédié à la modélisation de systèmes (si besoin au moyen d'extensions appelées *profiles*), ou encore XML Metadata Interchange (XMI) qui est un format basé sur XML employé dans le cadre de la sérialisation d'objets conformes à MOF (typiquement, un modèle UML est encodable vers un fichier au format XMI).

Le concept du MDA est de distinguer trois types de modèles :

- les modèles indépendants d'une plateforme, Platform Independant Model (PIM),
- les modèles de plateformes, Platform Description Model (PDM),
- les modèles spécifiques à une plateforme, Platform Specific Model (PSM).

La modélisation en MDA consiste à concevoir un PIM, c'est-a-dire un modèle faisant abstraction de tout langage et de tout détail d'implémentation. Le langage de modélisation UML, standard inclus dans l'approche MDA, est largement utilisé à cet effet. L'objectif est ensuite de parvenir à un PSM, donc un modèle prenant cette fois-ci en compte les éléments lié à la plateforme d'exécution (matériel, système d'exploitation, langage, etc.). La transformation d'un PIM en PSM s'effectue au moyen d'un PDM – donc un modèle

10. Par exemple en générant des balises en commentaires dans un code source, indiquant au développeur que seules les parties hors balises sont à sa disposition.

décrivant les élément d'exécution relatifs à une plateforme donnée – et d'un ensemble de règles de mappage et de transformation.

2.3 MDE pour la conception de SoC

Bien que l'approche MDA soit employée depuis ses débuts avec succès au sein d'équipes de développement de logiciels, c'est principalement après l'arrivée des spécifications UML 2.0 qu'on la trouve utilisée dans le domaine de la conception matérielle.

En effet, grâce à sa capacité d'extension via des *profils*, UML est employé et adapté dans de nombreux travaux adressant la conception de SoC, et de SoC reconfigurables en particulier.

2.3.1 Profils UML

UML est le principal langage de modélisation visuel de l'approche MDE, standardisé en 1997 par l'OMG, faisant alors partie intégrante de l'implémentation standard MDA. Répondant à des besoins de spécification, de communication et de documentation, UML a depuis été largement adopté tant académiquement qu'industriellement, favorisant ainsi la création d'un écosystème composé de nombreux outils le supportant [11].

Largement influencé par l'approche orientée objet, UML permet notamment de spécifier les aspects structuraux de systèmes via :

- des *classes*, descriptions des objets du systèmes (domaines de valeurs d'états, opérations fournies, etc.) ;
- des *composants*, éléments autonomes encapsulant des traitements et communiquant leurs exécutions via des *ports* définissant une interface ; un composant est remplaçable à l'exécution par toute *instance* de ce composant, c'est-à-dire toute implémentation disposant de la même interface ;
- des *déploiements*, représentant l'utilisation de l'infrastructure physique par le système et la manière dont ses composants sont répartis.

UML permet en outre de s'intéresser à la modélisation comportementale du système, notamment via :

- des *activités*, traductions algorithmiques de cas d'utilisation du système ;
- des *interactions*, séquences d'échanges de messages, manières de communiquer, entre des objets du systèmes, contraintes temporelles, etc. ;
- des *machines à état*, décrivant les états du systèmes et les conditions de transition entre états ;

11. http ://en.wikipedia.org/wiki/List_of_UML_tools

– etc.

Cependant, l'expressivité et la précision d'UML ne sont pas toujours bien définis pour certains cas de spécification de systèmes particuliers. À cet effet, UML peut être étendu au moyen de *profils*, c'est-à-dire des modèles (définis en UML) contenant entre autre des *stéréotypes* et des *valeurs étiquetées* (ou *tagged values*) permettant respectivement de spécialiser des classes et de leur donner des attributs par défaut. Étendre ainsi arbitrairement UML revient à perdre son standard, mais permet d'ajouter la sémantique nécessaire à la spécification de systèmes particuliers. La création de profil permet d'utiliser UML comme un DSL, plus précisément comme un outil de Domain Specific Modeling.

Plusieurs profils UML ont été développés dans le but d'adresser la conception de Soc, ou plus généralement, la conception de systèmes temps-réel, certains étant même devenu des standards. On distingue deux catégories :

la première regroupe les profils orientés bas-niveau, s'intéressant à la modélisation de circuits et à la réalisation de ses composants. Leur but est de permettre la génération de code décrivant ces circuits à partir de diagrammes UML, revenant ainsi à utiliser UML comme un HDL de haut niveau. UML for SoC [52] et UML for SystemC [119] sont parmi ces profils.

La deuxième catégorie concerne les profils modélisant les systèmes d'un point de vue fonctionnel. Disposer de modèles abstraits, dédiés à des aspects bien distincts du système, est une aide précieuse non seulement pour la compréhension et la communication entre les concepteurs, mais également pour l'analyse du système aspect par aspect. Par exemple, une analyse d'ordonnancement nécessite d'assimiler le système en tant que collection de processeurs, mémoires, etc. Plusieurs profils UML, dont *UML for Schedulability, Performance and Time* (SPT) [51], rentrent dans cette deuxième catégorie en permettant notamment d'annoter les fonctionnalités des composants modélisés selon leur nature (ce qui permet de distinguer s'ils sont de type mémoire, ressource de calcul ou de communication, etc.).

2.3.2 UML/MARTE en particulier

Parmi les profils UML dédiés à la conception de systèmes temps-réel, MARTE [49], un standard de l'OMG signifiant *Modeling and Analysis of Real-Time Systems*, se distingue par sa capacité à donner des concepts précis pour la modélisation d'architectures matérielles en relation avec un SoC ; on le considère comme le remplacement et le raffinement de SPT. Le profil MARTE désigne en réalité un ensemble de profils, regroupés en quatre packages : *foundations*, *design model*, *analysis model* et *annexes* cf. figure 2.2.

Le package *foundations* encapsule l'ensemble des profils contenant les éléments de

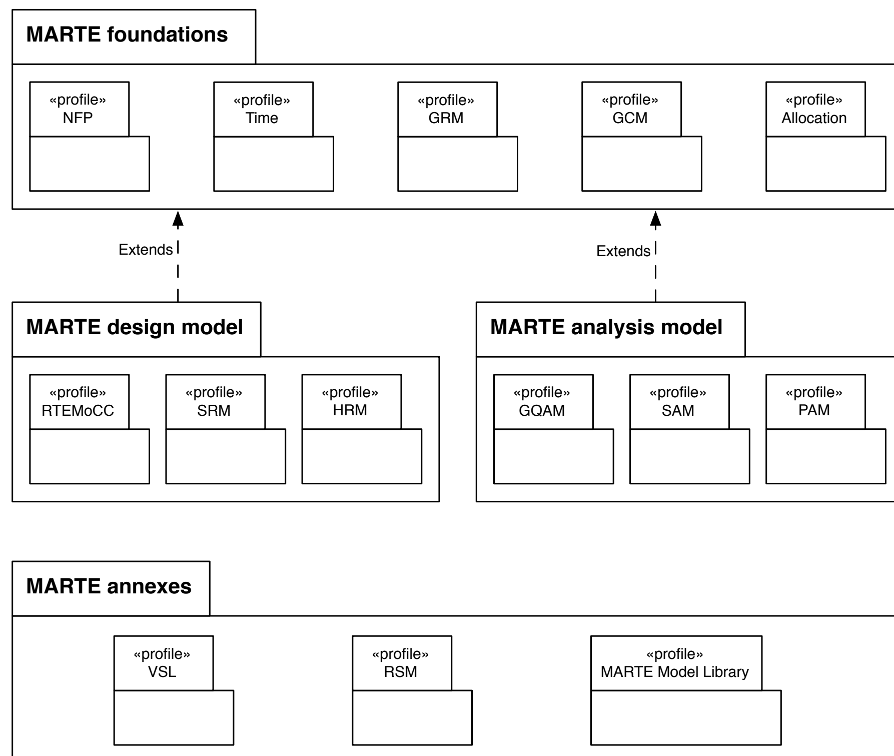


FIGURE 2.2 – Architecture globale du profil MARTE

noyau (*Core Elements*), décrivant les concepts transversaux de modélisation (modèles, éléments de modèles et comportements associés), tels que :

- les propriétés non fonctionnelles, *Non Functional Properties* (NFP), c'est à dire les informations, besoins, ou contraintes liées à l'implémentation (par exemple : consommation d'énergie, utilisation de mémoire, qualité de service, etc.) ;
- la modélisation temporelle, profil *Time*, pour représenter les contraintes de temps ;
- la modélisation de ressources, *Generic Resource Modeling* (GRM), telles que les ressources de calculs, de stockage, de synchronisation, etc. ;
- la modélisation de composants, *Generic Component Modeling* (GCM), désignant le concept de *composants*, *ports* et *instances* définis en 2.3.1 ;
- l'allocation de ressources, profil *Allocation*, permettant de spécifier la distribution spatiale et temporelle des éléments d'un modèle d'application vers des éléments d'un un modèle d'architecture ;

Le package *design model* spécialise le package *foundations* pour des aspects purement liés à la modélisation. Il regroupe :

- le modèle de calcul et de communication *Real-time and Embedded Models of Computation and Communication* RTEMoCC, donnant des concepts haut niveau pour la

modélisation quantitative de caractéristiques temps-réel (tailles de files de messages, deadlines, débit, etc.) ;
- la modélisation de ressources logicielles, *Software Resource Modeling* (SRM), afin de décrire des API pour un système d'exploitation temps-réel ;
- la modélisation de ressources matérielles, *Hardware Resource Modeling* (HRM), pour représenter les différentes vues (fonctionnelles, physiques) d'une architecture matérielle.

Le package *analysis model* spécialise également le package *foundations*, mais cette fois-ci en se focalisant sur la notion d'annotations de modèles réservées à l'analyse du système. On y distingue :
- la modélisation générique pour l'analyse quantitative, *Generic Quantitative Analysis Modeling* (GQAM), permettant de décrire comment le système utilise ses ressources (énergie, mémoire, etc.) afin de déterminer certaines mesure statistiques ;
- la modélisation de l'ordonnancement, *Schedulability Analysis Modeling* (SAM), étendant GQAM en le spécialisant pour l'analyse de scénarios d'ordonnancement ;
- la modélisation de la performance, *Performance Analysis Modeling* (PAM), spécialisant également GQAM pour l'analyse de propriétés temporelles.

Enfin le package *annexes* contient les profils et libraires prédéfinies permettant au concepteur d'ajouter des informations supplémentaires à ses modèles. Ce package distingue notamment :
- un langage dédié à la spécification de propriétés non fonctionnelles (NFP), *Value Specification Language* (VSL), permettant la description de types, paramètres, constantes, énumérations et expressions ;
- une sémantique opérationnelle pour la conception d'applications à traitements de données en parallèle, donnée par le profil *Repetitive Structure Modeling* (RSM) ;
- une librairie, MARTE Model Library, contenant les types primitifs employés dans MARTE ainsi que des types de données génériques et une librairie pour la notion de temps, définissant des types énumérés spécifiques au concept de temps.

[111] contient entre autre une étude comparative entre MARTE et plusieurs profils UML, ainsi que divers standards, notamment SysML – un langage de modélisation étendant un sous-ensemble des concepts UML – et AADL – un langage de description d'architecture pour les systèmes embarqués temps-réel majoritairement employé dans les domaines de l'avionique et de l'automobile.

Cette étude aboutit à la conclusion que MARTE surmonte systématiquement les limitations majeures des autres approches, par exemple grâce :
- à la modélisation des aspect non-fonctionnels (comparé à SysML) ;

- à l'allégement de la conception avec l'approche par composants UML (comparé à UML for SoC et UML for SystemC) ;
- à l'amélioration des possibilités d'analyse (ordonnancement, performance) via GQAM, SAM et PAM (comparé à SPT) ;
- à son emploi facilité dès le début de la conception (comparé à AADL).

L'étude note toutefois que MARTE, SysML et AADL peuvent être complémentaires sur certains aspects. Notamment SysML semble plus apte à la spécification des prérequis au plus tôt dans la conception, puis MARTE permet d'apporter les aspects non fonctionnels à des stades plus avancés. [102] Montre notamment la synergie pouvant exister en MARTE et SysML dans le cadre de la conception d'un système temps-réel. Concernant MARTE et AADL, une synergie intéressante existe dans le cas où des applications AADL sont modélisées en MARTE. Une telle méthodologie profite ainsi de l'approche de conception en amont fournie par MARTE, et de l'écosystème – techniques et outils de vérification et validation – de AADL. Ce principe est notamment décrit dans [39] et [120].

2.3.3 UML/MARTE pour la modélisation des SoC reconfigurables

MARTE offre clairement des nombreux avantages sur les autres approches, et les travaux utilisant ce profil pour la conception de SoC sont nombreux. En revanche, peu d'entre eux adressent la modélisation des SoC reconfigurables, particulièrement ceux disposant de la capacité de reconfiguration dynamique et partielle, et pour cause, la sémantique standard de MARTE n'est pas complète à ce sujet. C'est pourquoi divers travaux sont en cours dans le but d'adapter MARTE en conséquence.

Gaspard2

Gaspard2 (Graphical Array Specification for Parallel and Distributed) [41] est l'outil autour duquel gravite la majeure partie de ces travaux particuliers. Gaspard2 est un framework de conception de SoC orienté MDE. Bien qu'il utilise un sous-ensemble étendu de MARTE, Gaspard2 a fortement contribué au développement du profil. Le profil MARTE RSM par exemple, avec son modèle de calcul associé Array-OL [43], est directement inspiré de Gaspard2. Les profils HRM et Allocation ont également profité de certains aspects déjà présent dans cet outil.

Parmi ces travaux, [112] est le plus récent. Il porte notamment sur les avantages de MARTE en tant qu'outil pour la modélisation des configurations d'un SoC reconfigurable. Puis il discute en deux remarques majeures de ses limitations et ambiguïtés, particulièrement dues au manque de sémantique formelle dans le processus de contrôle de la reconfiguration.

La première remarque vient de l'absence de notion de délais de reconfiguration ; en effet lorsqu'un changement de configuration s'effectue au niveau architectural, l'architecture en question passe par une phase de stabilisation pendant laquelle les éléments en cours de modification ne doivent être utilisés. Les auteurs indiquent qu'une sémantique formelle de transition, telle que celle définie dans les automates de mode synchrones [92], doit être proposée.

La deuxième remarque expose le problème du contrôle de la reconfiguration à plusieurs niveaux. Dans un système complexe, plusieurs contrôleurs s'exécutant en parallèle et gérant différents aspects de reconfiguration peuvent être amenés à communiquer. Un comportement typique et difficile à gérer est celui de multiples reconfigurations demandées au même instant par différents contrôleurs. Les auteurs précisent que sans sémantique formelle de ce point de vue, des conflits peuvent se produire si les reconfigurations sont incompatibles.

Les auteurs adressent ensuite ses points difficiles en utilisant la sémantique de Gaspard2 pour améliorer leurs modèles MARTE, en employant notamment l'approche synchrone des automates de modes implémentés dans Gaspard2.

Cependant, du fait de la composition parallèle de leurs contrôleurs, spécifiés en tant qu'automates, il est intuitif d'envisager une explosion combinatoire quant à la gestion des reconfigurations possibles (le produit synchrone de deux automates étant exponentiel vis-à-vis de leurs tailles). La spécification correcte du contrôle est alors un problème complexe dans ce framework.

MOPCOM

Une autre approche récente, définie dans le projet MOPCOM [85], emploie MARTE pour la modélisation d'applications dynamiques ciblant des plateformes reconfigurables. Elle part du principe que les concepts de MARTE sont suffisamment larges pour capturer les besoins de modélisation des systèmes temps-réel embarqués et faciliter son intégration avec d'autres outils de conception (notamment la suite d'outils Xilinx). Pour pallier les problèmes d'ambiguïté dans MARTE, le projet MOPCOM expose une méthodologie de conception apportant une sémantique clairement définie.

Suivant l'approche MDA, cette méthodologie dispose d'un niveau de modélisation détaillé, *Detailed Modeling Level* (DML), reprenant les concepts PIM, PM et PSM décrits en 2.2.3.0, cf. figure 2.3. Le PIM est conçu comme un ensemble de tâches (composants) échangeant des événements (via leurs ports). Le comportement d'une tâche étant notamment défini au moyen de machines à états. Le PM est constitué d'un ensemble de propriétés intellectuelles, *Intellectual Properties* (IP), également modélisées grâce à des composants UML connectés par des ports. Ce modèle emploie le profile MARTE HRM afin de don-

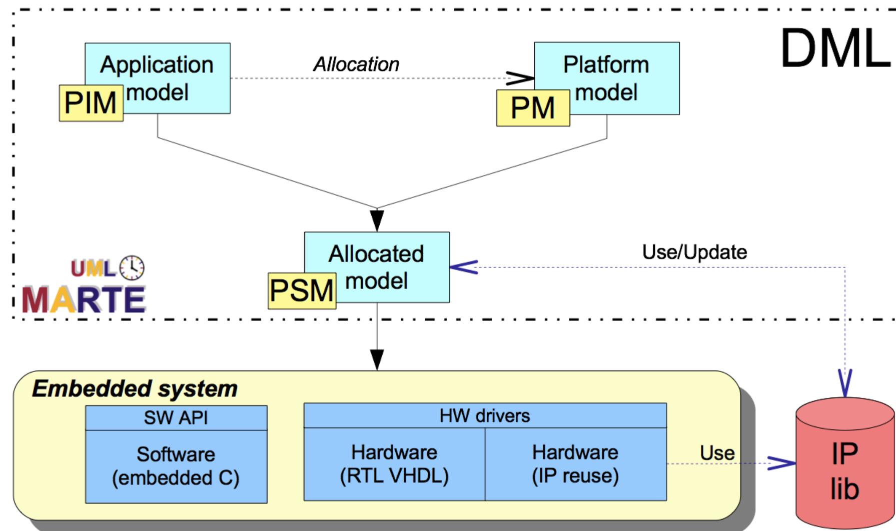


FIGURE 2.3 – Niveau de modélisation détaillé dans MOPCOM

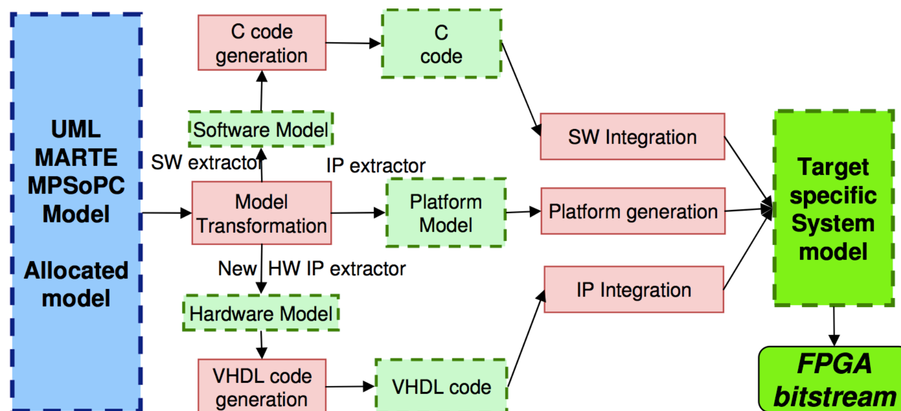


FIGURE 2.4 – Transformations de modèles dans MOPCOM

ner les caractéristiques des composants IP. Enfin chaque tâche du PIM est mappée à une ressource de calcul issue du PM, ce mappage étant effectué dans le PSM, appelé ici *Allocated Model*, au moyen de dépendances UML. C'est à-partir du PSM, modèle complet du système, qu'il est alors possible de procéder à la génération du code (C et VHDL) du système, ainsi que les fichiers de projet relatifs à l'intégration de la plateforme dans un flot de conception propriétaire (outils Xilinx), cf. figure 2.4.

Cette approche est intéressante dans le sens où seuls les éléments standards d'UML et MARTE sont utilisés et rendus non ambigus au moyen d'une méthodologie de conception. Cependant, n'exploitant pas les avancées non standardisées de Gaspard, la méthodologie MOPCOM ne profite pas de son outillage de vérification/validation et est moins apte à la conception d'application de traitement intensif du signal [111], domaine de prédilection de Gaspard2.

2.4 Conclusion

Ce chapitre à donné un aperçu des méthodes et outils de conception pour la réalisation de SoC. Des approches permettant de toujours repousser plus loin les niveaux d'abstraction vis-à-vis des détails d'implémentation apparaissent au fur et à mesure que les SoC se modernisent et deviennent plus complexes à concevoir.

Parmi ces approches, la modélisation s'est imposée – particulièrement depuis la standardisation du MDA – en tant que méthode d'abstraction de plus haut niveau. À tel point qu'il est possible de démarrer la conceptualisation d'un système en réalisant des modèles qui auront du sens par la suite dans le flot de développement, grâce à des transformations automatiques appropriées.

De par sa capacité à combler les manques des autres méthodes modélisation, le profil UML MARTE, standard du MDA, se distingue en tant qu'outil idéal – ou en tout cas hautement complémentaire (vis-à-vis de SysML et AADL notamment) – pour adresser la conception des SoC avec un haut niveau d'abstraction.

Cependant MARTE ne dispose pas encore de sémantique clairement définie pour la modélisation des SoC reconfigurables. À ce titre, le projet FAMOUS – dont l'objectif est de contribuer à MARTE – souhaite capitaliser sur les résultats obtenus dans le projet MOPCOM et les travaux autour de Gaspard2 afin de proposer respectivement une méthodologie de conception basée sur MARTE et une sémantique formelle pour le contrôle de la reconfiguration.

Le point particulier du contrôle de la reconfiguration est un problème difficile, spécialement lorsqu'il s'agit de prouver sa sûreté. C'est pourquoi une étude approfondie des travaux adressant la notion de contrôle d'architectures reconfigurables est nécessaire avant de proposer une contribution et une modélisation appropriée dans MARTE.

3

Contrôle de la reconfiguration

Sommaire

3.1 Introduction **57**
3.2 Notion de conception sûre **58**
3.2.1 Approches par test 58
3.2.2 Approches par méthodes formelles 61
3.3 Spécification du contrôle **66**
3.3.1 Systèmes réactifs synchrones 66
3.3.2 Synchronisme 68
3.3.3 Langages synchrones 69
3.3.4 D'autres approches de spécification du contrôle 73
3.4 Conclusion **74**

3.1 Introduction

Si la difficulté d'exprimer l'adaptativité des systèmes reconfigurables est notamment adressée par les techniques de modélisation, la sûreté de cette adaptativité – autrement dit le contrôle sûr de la reconfiguration – reste un point sensible qu'il convient d'adresser par des méthodes appropriées. Un des principaux défis dans la conception des systèmes reconfigurables est en effet d'assurer qu'il n'exhiberont pas, à l'exécution, de comportements non-désirés. Dans le cadre de certains systèmes, ignorer cet aspect peut même conduire à des catastrophes humaines, environnementales ou financières. La sûreté de conception est un domaine à part entière, faisant l'objet de nombreux travaux. Ce chapitre a pour objectif d'en donner un aperçu en traitant d'abord de la notion de sûreté dans le processus de conception d'un système, puis en s'intéressant à la spécification du contrôle en se focalisant sur les approches sûres.

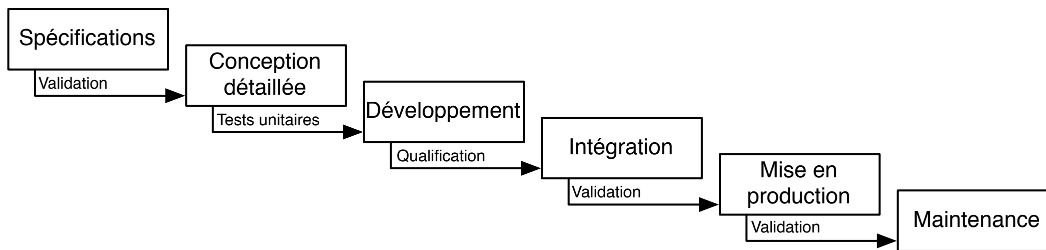


FIGURE 3.1 – Modèle en cascade

3.2 Notion de conception sûre

Deux approches visant à assurer la conception sont présentées dans cette section. Il s'agit des approches par test, largement répandues et employant l'expertise humaine pour vérifier les systèmes, et des approches formelles, avec lesquelles sont vérifiées automatiquement des propriétés d'exécution.

3.2.1 Approches par test

La livraison sur le marche d'un système logiciel ou matériel suppose généralement une phase coûteuse de tests. Dans le milieu de la conception matérielle, celle-ci est souvent conduite à l'aide d'outils de simulation appropriés, tels que ModelSim de Mentor Graphics.

Cette approche conventionnelle de validation d'un système réside dans l'élaboration – au sein du cycle de développement – de tests permettant de constater son bon fonctionnement ou de détecter des erreurs le cas échéant. La conception de systèmes matériels hérite notamment des méthodes employées à ce sujet dans le domaine du logiciel.

L'une des première approches historiques de validation par test de systèmes s'inspire d'une méthodologie issue de l'industrie du bâtiment : le modèle en cascade [121]. Le modèle en cascade (cf. figure 3.1) à pour principe de partir d'un ensemble de spécifications fonctionnelles, figées dès le début de la conception, puis de découper le projet en étapes clés sur un principe de non-retour avant de procéder à l'élaboration du système en effectuant des tests à chaque transition inter-étape. Lorsqu'une étape est achevée, elle est soumise à une revue approfondie : des experts de l'étape valident la correspondance entre ses objectifs et ses résultats. Si cette revue la juge satisfaisante, les résultats de l'étape servent de point d'entrée à la suivante.

Le défaut majeur de cette méthode vient du fait que la vérification du bon fonctionnement du système est réalisée trop tardivement, souvent lors de l'intégration, voire pendant la mise en production, les éventuels problèmes découverts à ces étapes causant généralement des coûts importants de correction. Par la suite, le modèle a été modifié

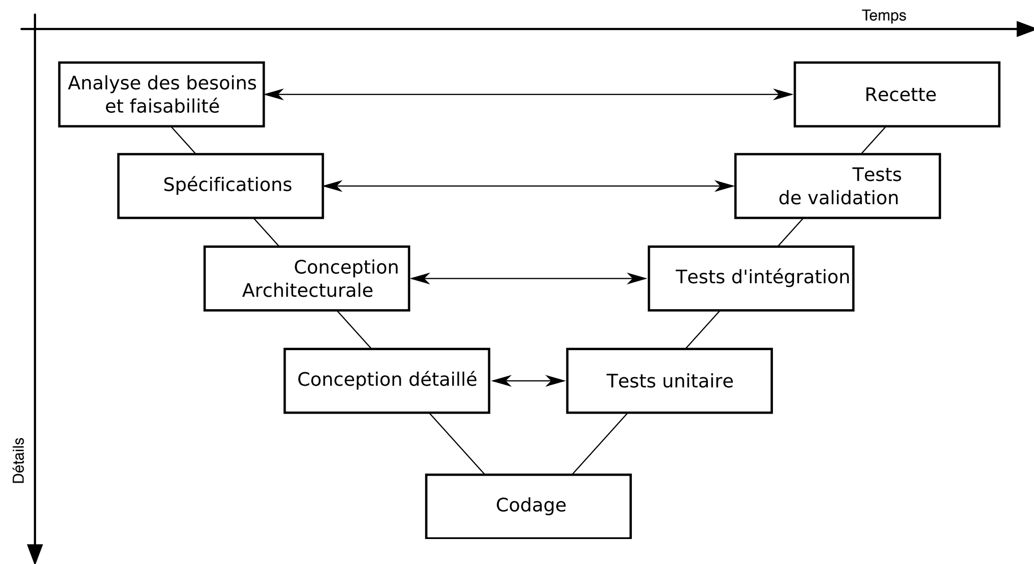


FIGURE 3.2 – Cycle en V

afin de permettre la remise en cause d'une étape précédente, mais dans la pratique, cette adaptation s'est avérée insuffisante.

Cette méthode trop rigide a conduit à l'établissement, au début des années 1980, du *Cycle en V* [94]. Ce cycle de conception (cf. figure 3.2) , qui s'est rapidement répandu dans l'industrie, permet de limiter les retours aux étapes précédentes en introduisant une partie de validation dite *montante*, symétrique à une partie de conception dite *descendante* ; chaque étape de la partie montante étant capable d'effectuer un retour sur l'étape descendante en vis-à-vis. Plus on descend dans ce schéma de conception, plus le niveau de conception est élevé, jusqu'à l'étape de codage effectif du système. Puis au fur et à mesure des étapes de tests, le niveau de détails diminue jusqu'à aboutir à la validation complète du système. Si une étape de test échoue, il n'est pas nécessaire de remonter jusqu'aux spécifications de départ (contrairement au modèle en cascade), mais seulement à l'étape de conception symétrique.

Le cycle en V bénéficie d'une large adoption industrielle et possède plusieurs implémentations standards dont l'une des plus connues est V-Modell [67], implémentation compatible avec plusieurs standards de qualité (dont ISO 9001 :2000, ISO/IEC 15288 et CMMI). V-Modell est par ailleurs employé dans la conception matérielle, notamment par EADS qui présente dans [78] comment le processus de conception logicielle, matérielle et logistique est intégré dans la méthodologie. [99] montre également de manière très détaillée son application dans un contexte de conception de services robotiques.

Bien qu'il permette une plus grande réactivité comparé au modèle en cascade, le cycle en V souffre toujours du problème de vérification trop tardive du système. Dans ce type de

méthodologie, le système est classiquement décomposé en parties développées séparément et intégrées seulement à la fin du processus de conception/développement, ce qui favorise un phénomène problématique : *l'effet tunnel*. L'effet tunnel est courant dans ces approches linéaires : les besoins du client sont captés seulement au début de la conception au moyen d'un cahier des charges, puis le système complet est développé en conformité et livré au client ; or souvent, à l'étape de livraison, les besoins du client ont évolué [12] et le système conçu n'est plus en phase avec ses attentes.

Pour pallier ce problème, des méthodes incrémentales dites *agiles* se sont développées : leur principe est qu'un noyau minimaliste du système est créé et validé, puis amélioré de manière incrémentale, chaque incrément étant immédiatement validé et intégré au noyau. Cela a pour avantages de rendre chaque développement d'incrément moins complexe que le précédent, d'intégrer progressivement les améliorations et de livrer voire mettre en service chaque incrément dès sa validation, ce qui fait intervenir le client tout au long du projet et non plus seulement au début et à la fin.

L'extreme programming [9] est l'une des implémentations les plus modernes et répandues de cette approche. Elle encapsule notamment le concept de développement piloté par les tests, *Test Driven Development* (TDD) [10], dans lequel les tests unitaires [13] ont ici la particularité d'être mis en œuvre avant le développement du système. L'objectif étant de coder suffisamment pour faire passer les tests les uns après les autres, jusqu'à ce que tous soient corrects.

Faisant intégralement partie du processus de développement, ces tests permettent au concepteur d'évaluer régulièrement la qualité du travail accompli, favorisant ainsi la détection et la correction au plus tôt des erreurs de conception, palliant alors l'effet tunnel inhérent aux approches linéaires. D'autre part, tout incrément nécessitant la réalisation de tests avant même d'être développé, aucune fonctionnalité du système n'est supposée échapper à la validation, ce qui est un gage de qualité majeur.

Largement employée dans le milieu du logiciel, le TDD (et son écosystème d'outils, ainsi que les méthodes agiles en général) s'immisce également dans la conception matérielle et notamment la création de systèmes embarqués [33].

La qualité des approches par test se renforce continuellement avec la modernisation des méthodes de conception. Mais si tester/simuler – sur un certain nombre de cas sensibles – l'exécution d'un système afin de s'assurer que son comportement est conforme à sa spécification est un aspect important voire incontournable dans le processus de validation, cela n'est pas suffisant pour prouver réellement sa sûreté de manière exhaustive. En effet,

12. Son métier a changé, les technologies se sont modernisées, voire il se rend compte de ses besoins réels.

13. Les tests permettant de s'assurer du fonctionnement correct d'une partie déterminée d'un système.

même si les tests révèlent une bonne part des erreurs – notamment grâce à l'assistance d'outils d'analyse fournissant divers indicateurs de qualité [14] –, ils ne constituent en rien une garantie : accumuler les présomptions n'est pas *prouver* au sens formel du terme.

3.2.2 Approches par méthodes formelles

La complexité croissante des systèmes embarqués – et des applications informatiques en général – couplée à leur présence quasi-ubiquitaire a malheureusement pour corollaire l'apparition de fonctionnements non-prévus aux conséquences parfois irréparables.

La littérature traitant de la sécurité et de la validité des systèmes destinés aux applications les plus sensibles fait généralement un rappel des faits les plus connus :

- L'exemple du Therac-25, un système informatique de radiothérapie, est l'un des plus importants du domaine médical : entre 1985 et 1987, six accidents dus à des surdoses de radiations – dont cinq ont entraîné la mort – ont impliqué l'usage de cet appareil. Le logiciel de contrôle du Therac-25 ne détectait pas l'anomalie de fonctionnement capable d'engendrer une irradiation massive. Ce contrôleur, programmé en assembleur – langage courant à l'époque – était difficile à analyser, et les concepteurs avaient négligé des étapes clés de sa validation. Ce dysfonctionnement est catégorisé comme le bug informatique le plus grave de l'histoire [15] [88].
- Un autre exemple est celui de l'échec du vol inaugural de la fusée Ariane 5 en 1996. Le contrôleur de vol [16], utilisé avec succès sur Ariane 4, n'avait pas été formellement validé pour supporter l'accélération du nouveau lanceur, 5 fois supérieure à celle d'Ariane 4. Une exception a été levée pendant la conversion de cette valeur d'accélération depuis une représentation en virgule flottante sur 64 bits vers un entier signé sur 16 bits, soit un espace insuffisant pour la stocker complètement [84]. Le contrôleur de vol, interprétant une valeur en vérité tronquée, provoqua l'instabilité de la fusée [17] qui s'autodétruisit quelques secondes après son décollage, causant la perte des trois satellites embarqués, estimés à 100 millions de dollars. Il s'agit du bug le plus coûteux répertorié.

Malgré le professionnalisme et l'expertise des concepteurs de ces systèmes [18], l'histoire est en effet riche d'exemples d'accidents dus à des fautes de conception. Or dans les domaines comportant notamment des risques pour la vie ou l'environnement (centrale nu-

14. Tels que la *couverture de code* décrivant le taux de code source testé dans un programme.
15. En tout cas, en tant que bug avéré et civil ; les bugs du domaine militaire étant rarement répertoriés publiquement.
16. Plus précisément sa centrale inertielle, soit le dispositif destiné à fournir les information de position, de vitesse, d'accélération et d'*attitude* (roulis, tangage et cap).
17. Braquage intempestif de ses moteurs.
18. Et parfois malgré les milliards de dollars investis, comme ce fût le cas pour Ariane 5.

cléaire, transport, appareils médicaux...), des risques financiers (transactions bancaires...), etc. la notion d'erreur est tout simplement interdite.

Les systèmes informatique concernés par ces domaines sont dit *critiques*, et leur vérification par des méthodes formelles est un moyen rigoureux de les certifier conformes d'après leurs spécifications. L'utilité des ces méthodes est indéniable, mais leur coût – en ressources humaines ou matérielles – est généralement important, c'est pourquoi leur amélioration [19] constitue un domaine de recherche particulièrement actif.

Le principe commun aux techniques de vérification formelles permettant de vérifier si un système S satisfait une propriété P – P représentant un ensemble de comportements souhaités – est d'établir, à l'aide d'une sémantique formelle, un modèle M du système S (par exemple sous forme d'automates de mode [92] ou par un système d'actions dans l'approche de Back [8]) et une description ϕ de P (par exemple en logique temporelle [106]), puis de chercher si ϕ est satisfait dans M, ce qui se note :

$$M \vDash \phi$$

Les sémantiques formelles de M et ϕ permettent de lever toute ambiguïté de la spécification informelle [20] du système telle qu'on la trouve dans le cahier des charges.

Deux techniques principales de vérification permettent de prouver formellement que $M \vDash \phi$, il s'agit de la *preuve de théorème* et de l'*analyse statique*.

Preuve de théorème

Dans l'approche par preuve de théorème, le principe est de poser un ensemble d'axiomes et de règles d'inférence puis de prouver un ensemble d'assertions – souvent à l'aide d'outils appelés *démonstrateurs de théorèmes* – afin d'aboutir à une conclusion du type $M \vDash \phi$. L'avantage d'une telle approche vient de sa concision : la technique de preuve est capable d'inférer des conclusions à partir de descriptions d'événements ou d'opérations permettant de faire évoluer le système ; en général, cette description des évolutions est petite comparée à la représentation explicite des états auxquels son application permet d'aboutir. Mais bien que l'on retrouve cette technique appliquée à la conception matérielle – notamment dans [107] qui fait usage de l'outil Isabelle [139] afin de prouver la génération correcte de placement de composants dans un circuit – elle reste peu employée en raison de son usage complexe. Même à l'aide d'outils, l'automatisation du processus est rarement possible et le concepteur est généralement obligé de guider la preuve en interagissant avec le démonstrateur de théorème [93].

19. Simplification d'emploi, élargissement du champ d'application, etc.
20. Exprimée par exemple en langage naturel.

Analyse statique

La deuxième approche, l'analyse statique de programmes, dont l'un des premiers outils aura été LINT [73] pour le langage C, emploie une variété de méthodes formelles capables de dériver automatiquement des informations comportementales d'un programme sans nécessiter son exécution. Une telle analyse est indécidable en raison du problème de l'arrêt [134] : il n'existe donc aucune méthode automatique capable de déterminer, d'après un programme, si celui-ci va produire ou non des erreurs à l'exécution. À partir de ce constat, il existe deux grandes familles d'analyse statique : l'*interprétation abstraite* et le *model-checking*.

L'interprétation abstraite [23] est une théorie de l'approximation discrète de sémantiques de systèmes informatiques principalement utilisée pour l'analyse et la vérification statique de logiciels. Elle réalise une exécution partielle d'un programme – mais contrôlée afin de terminer à coup sûr – en approximant son état de manière à aboutir à un point fixe d'exécution [21]. Schématiquement, l'approximation d'état correspond à un calcul symbolique d'un sur-ensemble des états accessibles du système. Si l'approximation trouvée au point fixe satisfait la propriété recherchée, alors le programme satisfait cette propriété de manière sûre pour tout exécution. Si ce n'est pas le cas en revanche, on ne sait pas [22] si la propriété est satisfaite, et on considère alors qu'elle ne l'est pas. Il y a en général un compromis à décider entre la précision de l'analyse et sa capacité à terminer. Dans [58], il est notamment fait usage de cette technique, en employant des intervalles de valeurs en terme d'abstraction de variables, afin d'optimiser les tailles de Look-Up Tables d'un circuit spécifié à l'aide de l'outil MADEO [81]. Cette même famille d'analyse statique est employée dans l'étude qui a suivi l'échec du vol inaugural d'Ariane 5 [80]. Celle-ci présente les moyens mis en œuvre pour vérifier formellement les éléments logiciels critiques du lanceur. La technique de preuve formelle est par ailleurs écartée en raison de la modification structurelle profonde qu'elle aurait nécessité sur le processus de développement : le modèle formel du programme à prouver aurait été trop difficile et coûteux à construire. L'analyse statique est par contre retenue, et appliquée avec succès grâce à l'outil IABC (INRIA Interprocedural Array Bounds Checker), afin de vérifier des propriétés d'exécution [23] concernant environ 80000 lignes de code ADA.

Le model-checking [113] [22] adresse le problème d'analyse statique d'une autre manière : au lieu de réaliser une abstraction d'un système, celui-ci doit être spécifié à l'aide d'une représentation finie d'états. Le model-checking considère en effet les systèmes à

21. L'état ne change plus lorsque le programme est exécuté à nouveau.
22. Justement en raison de l'approximation.
23. Initialisation correcte de variables, liste exhaustive des conflits potentiels d'accès à des variables partagées, liste exhaustive d'erreurs à l'exécution dues à la sémantique d'Ada.

états finis, ou qui peuvent être réduits vers des systèmes à états finis par abstraction. Une telle représentation permet d'analyser exhaustivement l'évolution du système lors de ses exécutions possibles. Le principe est donc de définir le modèle M du système à l'aide d'une structure à états finie (un graphe formé de nœuds et de transitions) telle que les automates de mode, ainsi qu'une propriété temporelle ϕ que l'on souhaite vraie pour tout exécution, et d'utiliser un outil de model-checking qui explorera M afin de vérifier qu'aucune évolution possible du système ne peut mettre en défaut la propriété ϕ. L'énumération explicite des états du système peut être coûteux, particulièrement lorsqu'il est représenté par plusieurs automates car il s'agit de combiner leurs états ce qui provoque une explosion combinatoire. C'est pourquoi les méthodes modernes de model-checking raisonnent sur des représentations symboliques [95] de ces états [24], dont l'une des plus connues étant le diagramme de décision binaire. CADP [70] et SIGALI [71] sont des exemples d'outils permettant d'effectuer la technique de model-checking de manière symbolique. SIGALI a par exemple été utilisé dans le cadre de la vérification du contrôle de reconfiguration dans une application de traitement intensif du signal spécifiée avec GASPARD2 [145]. CADP a notamment été expérimenté avec succès [40] pour vérifier les latences de communication dans un réseau sur puce, ou Network-on-Chip (NoC) [128].

Les méthodes de vérification présentées, qu'elles soient formelles ou non, ont un point commun : le modèle du système doit être intégralement spécifié par le concepteur avant de pouvoir procéder à la vérification d'une propriété d'exécution. Transposé à la conception du contrôle de reconfiguration, cela signifie que la représentation du contrôleur (son programme ou modèle) doit être complètement définie avant de pouvoir la tester, la prouver, ou l'analyser statiquement.

Cependant, une approche alternative consiste à synthétiser automatiquement – et formellement – la fonction de contrôle à partir de descriptions haut niveau. Si les techniques de vérification formelles ont principalement été développées par la communauté informatique, la synthèse formelle des fonctions de contrôle provient quant à elle de la communauté de l'automatique qui traite, entre autre, des systèmes dits *réactifs* s'exécutant sous contrôle d'une fonction appelée le *superviseur*. La théorie du contrôle par supervision est un cadre formel pour la vérification et la synthèse de superviseurs, la technique nous intéressant ici étant plus particulièrement celle de la *synthèse de contrôleur* (SdC).

Synthèse de contrôleur

Apparue dans les années 80 [115], la théorie du contrôle par supervision consiste à restreindre le comportement d'un système par le biais d'un superviseur de manière à ce

24. Un symbole représente un ensemble d'états

que le système ainsi contrôlé soit correct vis-à-vis d'un ensemble de propriétés (objectifs de contrôle) que le système initial ne vérifierait pas.

La SdC se base sur deux modèles : le modèle M du système à contrôler – typiquement spécifié à l'aide d'automates, comme c'est le cas pour l'outil de SdC SUPREMICA [3] – et un modèle ϕ du comportement désiré pour ce système, c'est-à-dire la propriété à assurer pour toute exécution. À partir de M et ϕ, un composant à intégrer au système, le *superviseur*, peut être synthétisé de manière automatique. Le superviseur a pour rôle de contrôler le système afin de le forcer à rester dans les limites imposées par la spécification ϕ, en interdisant dynamiquement au système de générer des événements capables de le compromettre. La contribution de la théorie du contrôle concerne la mise en place de la notion de *contrôlabilité*. En effet, dans la réalité, tous les événements ne peuvent être modifiés : certains d'entre eux sont dits *incontrôlables* – c'est par défaut le cas de tout événement provenant de l'environnement du système – et le superviseur ne doit jamais tenter de les modifier.

Dans la théorie du contrôle par supervision, pour un modèle M et une propriété ϕ donnés, un superviseur garantissant $M \vDash \phi$ n'existe que si M est *contrôlable*, cela signifie que M peut être forcé à rester dans les limites imposées par ϕ sans qu'il ait besoin de modifier les événements incontrôlables (pour toute exécution, et pour toutes valeurs d'incontrôlables). Il est également démontré que pour tout M et ϕ, si une solution de contrôle existe alors il existe toujours un unique superviseur optimal, garantissant que ϕ est respecté tout en permettant à M d'évoluer librement tant qu'il ne risque pas de se compromettre. L'optimalité désigne ici la restriction de M au strict nécessaire. Un tel superviseur est dit *maximalement permissif*, puisqu'il donne au système le maximum de liberté.

Dans [144], il est notamment fait usage de la SdC avec l'outil SIGALI, dans un contexte de contrôle d'architectures reconfigurables spécifiées dans GASPARD2. Les spécifications GASPARD2 sont transformées vers des représentations dite *synchrones*, à partir desquelles SIGALI est employé manuellement pour définir les propriétés désirées à l'exécution et effectuer une SdC. Mais comme précisé en substance dans [2] par les principaux contributeurs de SUPREMICA, la synthèse de contrôleur ne sera réellement acceptée dans l'industrie que lorsque des outils et méthodologies intégrées et *user friendly* seront développées. Typiquement, le processus de SdC devrait idéalement être masqué dans le processus de modélisation (et de transformation) au lieu d'être une étape de spécification supplémentaire.

De par leur capacité à raisonner sur des structures de types automates – soit une représentation proche de ce qui peut être spécifié en UML (et MARTE) grâce à des machines à états UML – les techniques de model-checking et de synthèse de contrôleur

semblent être particulièrement intéressantes afin d'obtenir un contrôle sûr. Cependant, bien que les concepteurs de systèmes de contrôle [25] soient souvent des experts de leur domaine d'application, ils ne sont pas nécessairement spécialistes des méthodes formelles. C'est pourquoi, avant d'adresser la modélisation formelle du contrôle et sa vérification, il convient d'analyser les méthodes classiquement employées pour sa spécification.

3.3 Spécification du contrôle

En raison de sa simplicité et de sa rigueur reposant sur des bases formelles bien définies, l'approche synchrone a séduit en particulier les industries développant des systèmes critiques en proposant notamment une solution originale à la programmation des systèmes embarqués, basée sur une abstraction du temps *réel* en un temps *logique*.

Cette approche a permis, depuis plus de deux décennies, de définir des langages expressifs de haut niveau, couplés à des sémantiques formelles et à des outils d'analyse dont le model-checking fait partie ; ces sémantiques et outils permettant d'améliorer respectivement l'intelligibilité et la sécurité du système conçu.

Cette section traite en priorité de l'approche synchrone en général, et de son utilisation dans le cadre de la spécification du contrôle

3.3.1 Systèmes réactifs synchrones

L'appellation générique des systèmes considérés dans l'approche synchrone est celle des *systèmes réactifs synchrones*. Un système réactif [62] se distingue des systèmes *transformationnels* et des systèmes *interactifs* :

- Un système transformationnel (cf. algorithme 1) a pour principe d'être initialisé avec des paramètres d'entrée (*input*), d'effectuer un calcul d'après ces paramètres (fonction f), et de produire le résultat (*output*) de ce calcul en sortie à la fin de son exécution. La sortie est donc une transformation de l'entrée. Un compilateur est un exemple typique de système transformationnel.
- Un système interactif (cf. algorithme 2) a pour particularité le fait d'être en interaction permanente avec son environnement. Ce type de système réagit à des événements qui déclenchent un calcul dont le résultat est retourné en sortie mais, à la différence d'un système transformationnel, sans provoquer la fin de l'exécution : ce principe étant répété en boucle infinie. Le temps de réaction peut être optimisé, mais n'est pas borné : le rythme du système est donné par la durée du calcul entre deux entrées successives, cette durée pouvant être arbitrairement longue ; le système

25. Et les concepteurs de systèmes en général.

n'acceptant de nouvelles entrées qu'à la fin de ce calcul. Les systèmes d'exploitation sont des exemples relevant de cette catégorie.

– Enfin, un système réactif (représenté graphiquement en figure 3.3) est identique à un système interactif, sa particularité étant son temps de réaction borné : il réagit à son environnement suivant une vitesse imposée par celui-ci. La durée de la réaction d'un tel système est assurée [26] de toujours terminer entre deux entrées successives. Lorsque les entrées sont du domaine continu, elles sont alors échantillonnées à un rythme pertinent vis-à-vis du système et de sa criticité. L'exécution d'un tel système est représentable sous la forme de l'algorithme 3 où une fonction transformationnelle f exécute un calcul à partir des entrées données au système et de son état courant. Ce calcul met l'état du système à jour et retourne son résultat en sortie. Cette étape est répétée dans une boucle infinie, activée par un événement – noté *tick* – représentant soit le rythme d'échantillonnage des entrées, soit l'arrivée d'une entrée si le système répond aux événements de l'environnement. L'exemple d'un système de contrôle-commande d'un avion (cf. figure 3.4) correspond à la définition d'un système réactif.

```
1 begin
2 |   Read(input);
3 |   output ← f(input);
4 |   Write(output);
5 end
```

Algorithme 1: Exécution d'un système transformationnel.

```
1 begin
2 |   while true do
3 |   |   Attente d'activation par l'environnement;
4 |   |   Read(input);
5 |   |   output ← f(input);
6 |   |   Write(output);
7 |   end
8 end
```

Algorithme 2: Exécution d'un système interactif.

26. Validée, prouvée.

```
1 begin
2     state ← initial ;
3     foreach tick do
4         Read(input);
5         (output,state) ← f(input, state);
6         Write(output);
7     end
8 end
```

Algorithme 3: Exécution d'un système reactif.

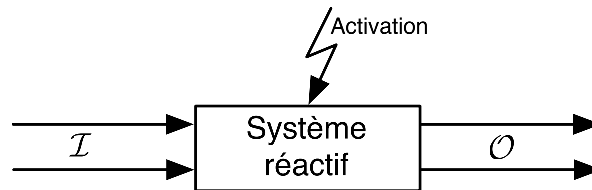


FIGURE 3.3 – Représentation visuelle d'un système réactif qui, à chaque activation par l'environnement, lit des entrées ($\mathcal{I}$), les traite en mettant son état à jour, et produit un résultat ($\mathcal{O}$).

Mission	- Stabiliser un avion naturellement instable. Calcul des lois de commandes et asservissement des gouvernes.
Entrées	- État et assiette de l'avion, plus les ordres du pilote.
Sorties	- Ordres de gouvernes et éventuelles alarmes.
Réactivité	- Boucle infinie dont le temps d'itération maximum est fixé à 1 milliseconde.

FIGURE 3.4 – Système de contrôle-commande d'un avion

3.3.2 Synchronisme

Dans l'interprétation synchrone de la réactivité [59], chaque instant de réaction considère l'ensemble des présences et valeurs (si présentes) des événements spécifiés en entrée.À noter qu'il existe d'autres interprétations possibles non strictement synchrones telles que StateCharts et StateMate [61], ou même asynchrones telles que Promela [87]. L'inconvénient des approches non strictement synchrones, notamment relevé dans [28], vient de la difficulté de leur composition en parallèle : il est nécessaire de définir un ordre de réaction entre les sous-systèmes composés, ce qui est abstrait dans l'approche synchrone. En effet, la composition synchrone est le point central de l'approche synchrone, cette dernière

désignant un ensemble de langages ainsi que leur compilation, mise en œuvre, analyse et vérification.

La composition synchrone est une opération composant deux processus définis par leur comportement. L'émission et la réception d'événements de deux processus ainsi composés sont considérées dans le même instant de réaction, c'est-à-dire dans le même instant logique vis-à-vis des entrées. Il s'agit bien sûr d'une abstraction, d'une image intuitive aidant à la compréhension du modèle d'un système dont les communications inter-processus ne sont en réalité ni instantanées, ni de durées nulles.

Formellement, un système réactif synchrone est donc un système sur lequel les trois abstractions suivantes sont faites :

- il est muni d'une échelle de temps discrète ;
- chaque réaction (activation par l'environnement ou échantillon si celui-ci est continu) est atomique et se déroule en un instant discret, les opérations effectives d'une réaction étant indissociables et non ordonnées ;
- toute réaction d'une partie du système ou de l'environnement est propagée à l'ensemble du système au même instant (propriété de la composition synchrone).

3.3.3 Langages synchrones

Les langages synchrones permettent de spécifier de tels systèmes grâce à ces abstractions. Si la première est triviale – un système numérique étant par nature soit discret, soit une discrétisation d'un système continu –, la seconde nécessite d'assurer en amont – par des méthodes et outils dédiés à l'analyse de contraintes temps-réel – que toute réaction possible respecte la cadence imposée. Si cet aspect temps-réel peut être vérifié, alors l'abstraction considérant le temps de réaction comme nul (instantané) est valide. Enfin, pour que la dernière abstraction soit celle d'une réalité concrète, une analyse de causalité est effectuée statiquement – c'est-à-dire pendant le processus de compilation de la spécification synchrone – afin de déterminer un ordre d'évaluation effectif des opérations de chaque instant.

Ces trois abstractions permettent ainsi la conception de systèmes réactifs déterministes et concurrents. Le modèle de temps discret, implicitement intégré à tout programme synchrone, est également une sémantique formelle très pratique : elle permet de simuler et vérifier une spécification indépendamment de son contexte réel d'exécution. C'est notamment la raison principale de l'adoption des langages synchrones dans le cadre de la programmation de systèmes embarqués critiques.

Les langages synchrones sont historiquement classés, suivant leur style de programmation, en deux familles : les langages impératifs (textuels ou graphiques), et les langages

déclaratifs (à caractère fonctionnel) ; leur point commun étant de disposer d'une opération de composition synchrone.

Les langages synchrones impératifs permettent de décrire explicitement un système réactif sous la forme d'un système de transitions, dans lequel l'état du système est explicite. Le langage Esterel [14], permet de décrire des processus au moyen d'instructions impératives : boucles, exceptions, structures de contrôle (séquentielles, parallèles ou conditionnelles) munies de points d'attente de l'occurence d'événements ou de signaux (événements valués). Il dispose notamment de structures préemptives permettant d'interrompre un sous-programme. Les langages SyncCharts [7] et Argos [91] sont des représentants graphiques de cette famille, à base de graphes à états concurrents hiérarchiques. Ils permettent de spécifier le système à partir de machines de Mealy [97] généralisées (leurs types de données peuvent être autres que booléens).

Dans le langages synchrones déclaratifs, la sémantique s'exprime en termes de *flots de données* : les valeurs portées par l'échelle de temps discret définissant le système sont considérées comme des séquences infinies de valeurs, ou *flots*. À chaque instant discret, la relation entre les valeurs d'entrées et celles de sorties est définie par une représentation équationnelle entre les flots : un programme spécifié dans un tel langage est assimilable à un système d'équations, dont la sémantique est qu'un flot d'entrées donne lieu à l'exécution de calculs définis dans les opérateurs, ceux-ci produisant des résultats soit consommés par d'autres opérateurs, soit retournés en sortie du programme. Les équations sont évaluées en concurrence dans un instant donné, et non en séquence, conformément à l'abstraction synchrone. Leur ordre réel d'évaluation étant déterminé à la compilation à partir de leurs inter-dépendances. À noter que sans dépendance particulière entre deux équations, un ordre arbitraire est alors choisi à la compilation afin de permettre une exécution déterministe.

L'évolution d'un système synchrone dans le temps s'exprime particulièrement bien dans la sémantique flots de données de ces langages. Par exemple, si x et y sont deux flots de données tel que $x = x_0, x_1, ...$ et $y = y_0, y_1, ...$ (l'indice des valeurs correspondant à l'instant logique auquel elles sont affectées), l'évolution temporelle de y donnée par la somme des valeurs passées de x s'exprime par le système d'équations suivant :

$$\begin{cases} y_0 & = & x_0 \\ y_t & = & y_{t-1} + x_t \quad \text{si } t \geq 1 \end{cases}$$

Chaque langage synchrone déclaratif dispose d'une syntaxe pour implémenter de manière synthétique des systèmes spécifiables sous forme d'équations mathématiques tels que ci-dessus. Par exemple, en HEPTAGON/BZR [29], ce programme s'écrit :

```
y = x -> pre(y) + x;
```

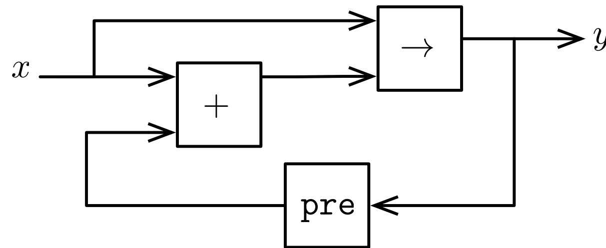


FIGURE 3.5 – Traduction du programme HEPTAGON/BZR (`y = x -> pre(y) + x;`) en réseau de Kahn.

La syntaxe est également optimisée pour la spécification de systèmes de types diagrammes de blocs. Un programme flot de données peut notamment être considéré comme un réseau de Kahn, c'est-à-dire un réseau de processus communiquant par files d'attentes et exhibant un comportement déterministe décorrélé de tout temps de calcul ou de communication [74]. Dans un programme à la fois synchrone et causal (sans dépendance circulaire parmi ses équations), il existe un ordonnancement pour lequel les files d'attente du réseau de Kahn correspondant sont nulles. La figure 3.5 montre la traduction en réseau de Kahn de l'exemple de programme précédent.

Le langage Heptagon, ainsi que BZR, LUSTRE [60], LUCID SYNCHRONE [19] et SIGNAL [83] sont des langages synchrones de type déclaratif. LUSTRE a une approche fonctionnelle, dans le sens où un programme LUSTRE est défini par une fonction des flots d'entrées vers ceux de sorties, dont les opérateurs sont structurés en nœuds composés d'instructions temporelles (telles que `pre` – qui vient d'être aperçu en Heptagon/BZR – permettant d'accéder à la valeur précédente d'un signal). LUCID SYNCHRONE est une extension de LUSTRE en tant que langage fonctionnel d'ordre supérieur. SIGNAL a quant à lui une approche relationnelle : une équation est une relation entre les flots, et non une définition d'un flot en fonction d'autres flots. Une telle relation peut donner lieu à diverses sorties possibles pour un flot d'entrée, ou divers flots internes pour une entrée et une sortie données. Cette spécification est donc non-déterministe, et le style de programmation est dit *par contraintes* : un ensemble de comportements possibles, défini par un ensemble d'équations, est réduit progressivement à la compilation jusqu'à obtenir une éventuelle solution fonctionnelle à partir de laquelle une version déterministe peut-être produite.

Les deux familles, impératives et déclaratives, offrent des moyens et outils complémentaires pour la programmation de systèmes réactifs. Lorsque le système est *orienté flots de données* (systèmes de régulation, traitement du signal, etc.), le formalisme de diagramme de blocs des langages synchrones déclaratifs est plus adapté. Au contraire, si le système est plutôt *orienté contrôle* (pilotes, systèmes à événements discrets, etc.), les langages impératifs synchrones sont un meilleur choix.

Cependant, la plupart des systèmes réels n'appartient pas à une catégorie particulière et mélange souvent les deux styles, ce qui a conduit à la proposition de solutions multi-paradigmes. Par exemple, l'extension SignalGTi [122] de Signal introduit la notion de tâches, avec intervalles de temps, préemption, etc. dans son paradigme flot de données. Le langage des automates de modes [90], généralisé dans Lucid Synchrone, est un autre exemple d'intégration de fonctionnalités impératives (ici la capacité à décrire un système par des machines à états) dans un langage déclaratif.

Enfin BZR, le plus récent de ces langages, est issu d'Heptagon qui est un autre langage synchrone déclaratif incorporant le langage des automates de mode [27]. Mais son principal atout par rapport aux autres langages réside dans son fort couplage avec l'outil de vérification formel Sigali, qu'il rend simple à utiliser : BZR dispose en effet d'une syntaxe à base de *contrats* permettant l'implantation correcte par construction de composants, grâce à une application modulaire de la synthèse de contrôleur offerte par Sigali. BZR permet ainsi de spécifier un système dynamique dont les différents modules sont reconfigurables sur occurrence d'événements contrôlables et/ou non-contrôlables, puis de spécifier les propriétés d'exécution désirées de ces modules (contraintes de compositions, etc.) et d'obtenir un superviseur, c'est-à-dire un contrôleur de reconfiguration correspondant (s'il existe) garantissant ces propriétés par une valuation appropriée des événements contrôlables.

```
1 begin
2   state ← initial ;
3   foreach tick do
4     Read(I_u);
5     I_c ← ComputeControllables(I_u, state);
6     (output,state) ← f(I_u, I_c, state);
7     Write(output);
8   end
9 end
```

Algorithme 4: Exécution d'un système reactif sous contrôle.

L'exécution d'un programme BZR est celle d'un système réactif sous contrôle, dont le principe est donné dans l'algorithme 4. La partie contrôle (le superviseur), synthétisée lors de la compilation et symbolisée ici par la fonction `ComputeControllables`, s'insère dans la boucle d'exécution du système réactif. Ce modèle prend donc en entrée à chaque instant

27. Heptagon est de conception relativement proche de Lustre, mais destiné à l'investigation de la compilation modulaire des langages synchrones.

logique des événements de l'environnement notés $\mathcal{I}_u$ (non-contrôlables), puis demande au superviseur de calculer les valeurs des signaux contrôlables, $\mathcal{I}_c$, en fonction de l'état courant et des valeurs des événements non-contrôlables. Tous les signaux étant valués, la fonction transformationnelle f peut alors effectuer un calcul en fonction de leurs valeurs et de celle de l'état courant, mettant ensuite l'état du système à jour et retournant son résultat en sortie.

BZR combine les avantages de l'approche fonctionnelle et de l'approche impérative grâces aux automates de mode, tout en exploitant les aspects formels de SIGALI. Sa capacité à simplifier l'utilisation de ce cadre mathématique formel le rend particulièrement attractif. BZR a notamment été utilisé avec succès dans divers contextes où le contrôle automatisé est un élément clé dans la génération de code du système. Dans [17], il est notamment employé pour assurer l'administration efficace de ressources (énergie, charge réseau) dans un système multi-serveurs de gestion de communications.

Bien qu'il existe d'autres outils de synthèse de contrôleur – notamment SUPREMICA [3] dont les modèles sont basés sur des automates à états finis déterministes, ou STCT [148] employant des représentations intermédiaires à base de diagrammes de décision entiers (*Integer Decision Diagram*, IDD) – BZR et SIGALI constituent à notre connaissance le seul couplage existant, permettant de combiner les avantages des langages synchrones pour la modélisation de systèmes réactifs et de la synthèse de contrôleur pour la génération automatique et formelle du code de contrôle de reconfiguration.

3.3.4 D'autres approches de spécification du contrôle

L'approche synchrone qui vient d'être présentée, et son couplage avec des techniques formelles – notamment la synthèse de contrôleur –, adresse essentiellement la modélisation de systèmes réactifs discrets et leur contrôle. Des approches alternatives – encore issues de la communauté de l'automatique – concernent les systèmes continus ainsi que les systèmes hybrides (comportant des évolutions continues et des phénomènes discrets qui leur sont liés), élargissant le champ des possibilités de modélisation.

Les systèmes continus sont constitués d'éléments caractérisés par une mesure qui peut prendre une infinité de valeurs (température d'une pièce ou d'un objet, vitesse d'un mobile, niveau dans un réservoir, etc.). La gestion de ces systèmes fait appel à des outils mathématiques aptes à la représentation de la dynamique continue, notamment des équations différentielles assorties de diverses transformations (Laplace, Fourier...).

Les systèmes hybrides [147], font quant à eux intervenir explicitement et simultanément des phénomènes ou des modèles de type continu et événementiel. L'évolution au cours du temps de ces systèmes est décrite par un ensemble de lois mathématiques

continues (équations différentielles, etc.) soumises à des éléments décisionnels discrets ou événementiels.

Les systèmes multimédias – où des tâches coordonnant des données audio ou vidéo sont synchronisées sur l'occurrence d'événements – ou les systèmes de vision artificielle – délivrant des services suivant leur interprétation d'un flux vidéo – sont des exemples appartenant à la famille de systèmes hybrides. Le contrôle hybride d'un système robotique de vision artificiel est notamment adressé dans [123], avec l'emploi de SignalGTi.

Enfin, dans [31], une méthode à base de contrôle continu pour la gestion de l'auto-adaptation dynamique d'un système embarqué est présentée. Celle-ci est particulièrement intéressante dans le sens où elle soulève et adresse le problème des reconfigurations intempestives : la reconfiguration a un coût (temps, énergie) non négligeable dans un système embarqué, aussi sa régulation est une solution efficace si le système peut tolérer d'être dans une configuration sous-optimale pendant un certain temps.

Le problème des reconfigurations intempestives est transversal dans les systèmes dynamiquement reconfigurables. Mais bien qu'il s'agisse ici d'une solution en terme de contrôle continu, elle mérite d'être étudiée afin de la combiner avec les techniques discrètes formelles dans une approche hybride : l'objectif étant et de profiter de cette technique de régulation afin d'optimiser le contrôle de la reconfiguration, tout en conservant le système contrôlé dans un espace d'états spécifié par une propriété garantie par les techniques formelles.

3.4 Conclusion

Ce chapitre a présenté les diverses approches pour la conception sûre des systèmes, et notamment des systèmes embarqués reconfigurables. Les approches conventionnelles par test sont de loin les plus répandues industriellement, souvent pour une question de coût, mais aussi en raison d'un manque d'expertise couplé à un manque de méthodologies adaptées pour l'emploi des techniques formelles.

S'il est vrai que les techniques à base de tests sont généralement moins coûteuses à mettre en place, le coût risqué à la moindre erreur dans un système critique peut parfois être suffisamment élevé pour motiver l'emploi de méthodes capables de prouver automatiquement son comportement correct.

Parmi ces méthodes, le model-checking et la synthèse de contrôleur sont particulièrement intéressantes en raison de leur capacité à traiter des spécifications à base d'automates de modes, sémantiquement proches des machines à état UML. L'avantage de la synthèse de contrôleur par rapport à d'autres approches étant la possibilité de spécifier un système

par contraintes, simplifiant alors la conception du contrôle de reconfiguration dont le code peut être automatiquement et formellement obtenu.

Ce chapitre a également montré diverses manières de spécifier le contrôle de la reconfiguration. L'approche synchrone est particulièrement adaptée pour la modélisation sûre des systèmes reconfigurables, notamment grâce aux outils formels gravitant autour de sa sémantique (dont le model-checking et la synthèse de contrôleur). Les travaux autour des systèmes continus et hybrides montrent toutefois qu'il est possible d'élargir les possibilités de contrôle, notamment en terme d'optimisation. Cette étude propose donc de s'intéresser à la combinaison de l'approche synchrone pour la conception sûre du contrôle de reconfiguration, sans toutefois exclure les approches continues pour permettre la mise en œuvre d'éventuelles optimisations.

Notre intérêt pour la modélisation du contrôle sûr, élément clé de cette étude, ainsi que notre motivation pour la mise en œuvre d'un flot de conception approprié, ont été présentés la première fois dans [56].

Deuxième partie

Contribution et méthodologie

4

Modélisation du contrôle dans MARTE

Sommaire

4.1 Introduction **79**
4.2 Sémantique des élements standards **80**
4.2.1 Modélisation de l'application 81
4.2.2 Modélisation de l'architecture 88
4.2.3 Modélisation de l'allocation 88
4.2.4 Ajout de propriétés et contraintes 90
4.3 Extension du profil MARTE **93**
4.3.1 Composition générique des propriétés non fonctionnelles . . 94
4.3.2 Modélisation d'un contrôleur générique 96
4.4 Conclusion **100**

4.1 Introduction

Le choix de MARTE dans le projet FAMOUS est d'abord lié aux avantages procurés par UML. L'emploi d'UML dans le domaine du logiciel à permis de réduire les temps de développement et de simplifier la communication entre les différents concepteurs. La conception en parallèle d'éléments logiciels et matériels devenant récurrente dans la mise en œuvre de systèmes embarqués, UML a été étendu – notamment avec le profil MARTE – afin de répondre aux besoins de ce secteur.

MARTE est un des standards les plus importants pour la modélisation de ces systèmes, mais est encore loin d'être complet pour traiter ceux concernés par la notion de reconfiguration dynamique et partielle.

Le premier objectif du projet FAMOUS est d'améliorer la modélisation MARTE de systèmes embarqués dynamiquement reconfigurables. Cela concerne deux notions clés :

– la définition de modèles de systèmes reconfigurables, caractérisés par l'exécution de tâches concurrentes, disposant de plusieurs implémentations et mappables en tant tâches matérielles (co-processeur) sur un FPGA ou en tant que tâches logicielles sur un GPP.
– la mise en place d'un système de métriques, à des fins d'analyse, permettant de spécifier des propriétés non fonctionnelles (issues par exemple d'outils de profiling) aux différentes configurations du système.

Au sein de cet objectif, la présente étude propose de traiter la modélisation du contrôle de la reconfiguration. La notion de contrôle distingue ici le *choix* de configuration, par opposition à la *diffusion* de configuration lors de l'exécution, c'est-à-dire la gestion de reconfiguration d'un graphe de tâches, avec prise en charge des priorités de reconfigurations et des suspensions/reprises de tâches en cours d'exécution.

Deux aspects du flots FAMOUS sont ici simplifiés, afin de se focaliser uniquement sur l'aspect contrôle :

– l'aspect particulier de la diffusion, adressé par d'autres acteurs du projet FAMOUS, dont les premiers résultats sont disponibles dans [138].
– l'aspect de modélisation de la plateforme, qui fait l'objet d'une contribution approfondie dans [20] pour la modélisation haut-niveau dans MARTE, et dans [105] pour la transformation vers une représentation standard de composants reconfigurables basée sur IP-Xact.

Ce chapitre propose de définir un niveau d'abstraction pour la vérification de propriétés de contrôle. Il s'agit d'établir un modèle de tâches reconfigurables dont les implémentations se réalisent soit logiciellement soit matériellement ; ces implémentations doivent pouvoir disposer de propriétés non-fonctionnelles composables afin de permettre l'évaluation de la qualité d'une configuration (composition des propriétés de ses implémentations) ; enfin, une sémantique d'exécution sous-contrôle doit être proposée, faisant abstraction des aspects particuliers de la diffusion (suspension/reprise, séquencement, etc.) en considérant qu'une reconfiguration s'effectue dans un instant logique.

Ce chapitre capitalise en particulier sur les résultats obtenus dans [136] et [111] – mentionnés respectivement par *l'approche* MoPCoM et *l'approche* Gaspard– et apporte une évolution quant à la modélisation du contrôle de reconfiguration.

Les bases des concepts présentés ici ont notamment été publiées dans [137].

4.2 Sémantique des élements standards

Pour rappel, un modèle MARTE est composé de trois parties : le *Platform Independant Model* (PIM), le *Platform Model* (PM) et le *Platform Specific Model* (PSM).

Dans le PIM tout comme dans le PM, un ensemble d'éléments, définis par des composants UML communiquent ensemble via des ports. Ces composants sont respectivement des tâches et des ressources matérielles. Le PSM résulte de l'allocation – via des dépendances UML – des éléments du PIM vers des éléments du PM ; c'est à partir d'un tel modèle qu'il est possible de générer, au moins en partie, le code du système.

La première étape, qui fait l'objet de cette section, consiste à rassembler et décrire la sémantique d'éléments MARTE standards, employés ici spécifiquement pour la définition d'un système composé de tâches reconfigurables, en prenant en compte l'objectif futur de transformation vers une représentation formelle pour sa vérification.

4.2.1 Modélisation de l'application

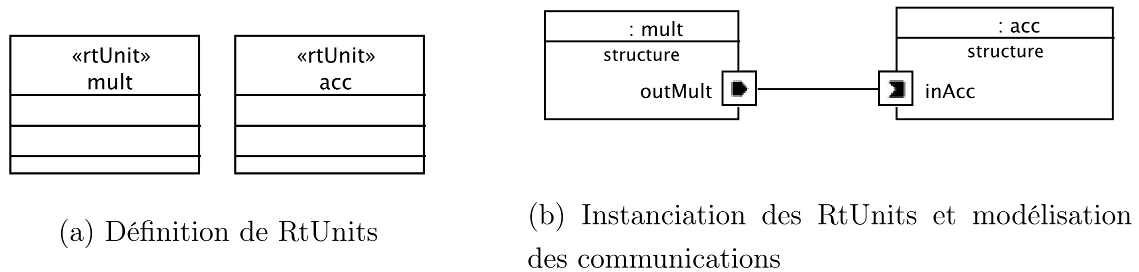
Spécification structurelle

L'élément clé de la modélisation d'application est le composant UML. Il définit une tâche indépendante, capable de communiquer via des ports avec d'autres tâches au moyen de connexion reliant leurs ports, ce qui forme ainsi un graphe. Le stéréotype «RtUnit» (du profil MARTE HLAM), appliqué à un composant UML, est la brique de base pour la spécification de propriétés temps-réel, soit la sémantique d'exécution d'une tâche. Dans l'approche MoPCoM, c'est notamment cette solution qui est retenue. Mais la communication – réception/émission d'événements – entre les tâches du modèle d'application (PIM) est réalisée de manière asynchrone. Or un tel choix ne peut être repris dans le cadre de FAMOUS, étant donné la nature synchrone des systèmes qui doivent être modélisés.

Dans Gaspard en revanche, le framework étant focalisé sur les traitements parallèles intensifs, les composants de l'application ne sont pas stéréotypés, mais disposent de ports stéréotypés par MARTE «FlowPort» permettant de spécifier des métadonnées définissant la manière de répartir (en parallèle) les communications entre tâches. Cette sémantique d'exécution, initiée dans Gaspard par le modèle de calcul Array-OL [43], se retrouve également dans MARTE avec le profil MARTE RSM.

Bien que cette approche soit spécifique, elle est cependant proche de ce qui est recherché dans FAMOUS : un graphe de tâches en Array-OL dispose d'une sémantique d'exécution à la fois synchrone et causale (pas de cycle).

L'approche FAMOUS ayant une portée plus générique, le principe de stéréotypage de composants UML en «RtUnit» est conservé, afin de permettre au concepteur d'ajouter ses propres informations temps-réel, et la communication s'effectue via des «FlowPort», autorisant la mise en place d'un modèle de calcul adapté à l'approche synchrone (réseau de Kahn [74], Array-OL, etc.).



(a) Définition de RtUnits

(b) Instanciation des RtUnits et modélisation des communications

FIGURE 4.1 – Principe de modélisation d'un graphe de tâches en MARTE

La figure 4.1a donne un exemple de définition de deux tâches, *mult* et *acc*, stéréotypées par «RtUnit», dont l'instantiation (en vue composant UML) est donnée en figure 4.1b, avec la modélisation des communications. Ces tâches sont exécutées en séquence, le sens de communication est donné par les propriétés de leurs ports : *outMult* et *inAcc* sont deux ports UML stéréotypés par «FlowPort» en MARTE ; la représentation graphique indique que l'attribut *direction* de ce stéréotype est valué à "out" pour *outMult* et à "in" pour *inAcc*, reflétant ainsi le sens en entrée (in) ou en sortie (out) du flot de donnée.

La sémantique de ces deux tâches spécifiées ainsi, sans plus de précision quant aux stéréotypes «RtUnit» et «FlowPort», est équivalente à celle d'un réseau de Kahn dont les files d'attente sont nulles, soit un modèle d'exécution admis dans l'approche synchrone.

Spécification comportementale

La représentation structurelle donnée précédemment ne définit que les interfaces des communications entre tâches. La spécification comportementale de ces tâches est une notion clé pour la spécification de la reconfiguration dynamique.

L'approche MoPCoM adresse deux cas de figures : les reconfigurations pour l'exploitation temporelle de ressources physiques au sein d'une séquence d'exécution de tâches, et les changements d'implémentations de tâches. La méthodologie MoPCoM n'imposant pas de modèle de calcul particulier, il est nécessaire de traiter le premier type de reconfiguration afin de définir l'ordre dans lequel chaque ressource physique impactée doit être modifiée pour supporter l'exécution correcte d'une séquence de tâches. La méthodologie propose dans ce cas d'employer une tâche («RtUnit») particulière, nommée contrôleur, responsable du flot de contrôle et de l'ordonnancement des tâches. Celle-ci reçoit des événements en provenance du système (émis par des capteurs, par d'autres tâches, etc.) et communique avec les autres tâches afin de déclencher leur reconfiguration quand cela est nécessaire dans la séquence d'ordonnancement. Le comportement du contrôleur est défini par le concepteur au moyen d'une classe encapsulée, disposant d'une machine à état UML. La figure 4.2 montre un tel contrôleur, *ctrl*, ainsi qu'une tâche pilotée nommée *adder* ; le

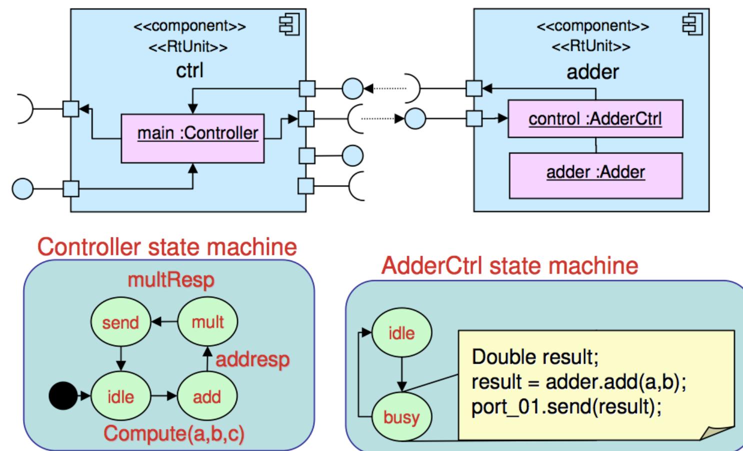


FIGURE 4.2 – Modélisation d'un contrôleur reponsable du flot de contrôle et de l'ordonnancement de tâches dans MOPCOM

comportement du contrôleur est implémenté au moyen d'une machine à état dans l'instance *main* de la classe *Controller* encapsulée dans *ctrl*, explicitant l'ordonnancement des tâches pilotées (add, puis mult, etc.) ; chaque tâche pilotée dispose, sur le même principe, d'une machine à état qui permet de l'activer ou de la suspendre.

En disposant d'un modèle de calcul approprié, par exemple un réseau de Kahn, une telle démarche de conception est alors dispensable car l'ordre de reconfiguration de tâches au sein d'une séquence est implicite : il suit l'ordonnancement imposé. L'approche GASPARD, reposant sur le modèle de calcul Array-OL, n'adresse d'ailleurs pas explicitement ce type de reconfiguration.

Concernant cette étude, l'ordonnancement des tâches est abstrait dans l'approche synchrone : une séquence s'exécutant dans un instant discret. Pour ce type de reconfiguration, il sera donc ici supposé soit trivial (si le modèle de calcul est par exemple un réseau de Kahn), soit adressé par l'acteur du projet FAMOUS traitant du principe de diffusion d'un ordre de reconfiguration.

Le deuxième cas de reconfiguration, désignant le changement d'implémentation d'une ou plusieurs tâches données entre deux séquences, est adressé dans MOPCOM de deux manières : les implémentations – représentées par des classes UML – d'une tâche («RtUnit») sont organisées soit dans un pattern *Strategy*, soit dans un pattern *State* (cf. Design Patterns [42]). Le pattern *Strategy* est adapté pour la sélection d'une configuration en la désignant via un identifiant, tandis que le pattern *State* est préférable pour la représentation de comportements évoluant sur occurrence d'événements ; ce dernier pattern nécessite donc d'être associé à une machine à états afin de représenter les changements d'états sur occurrence d'événements.

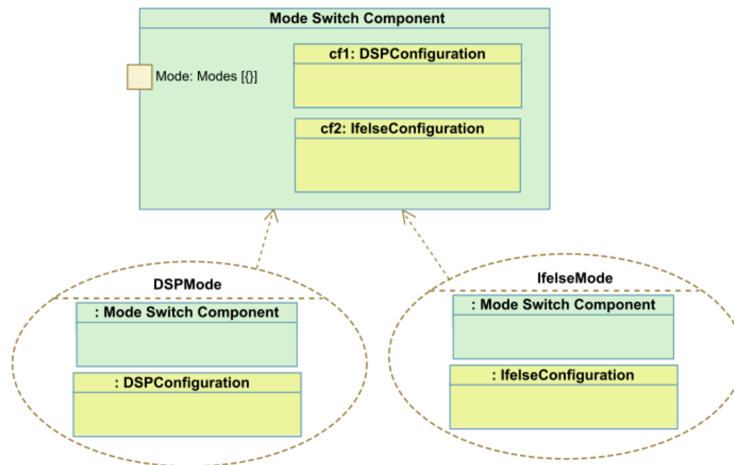
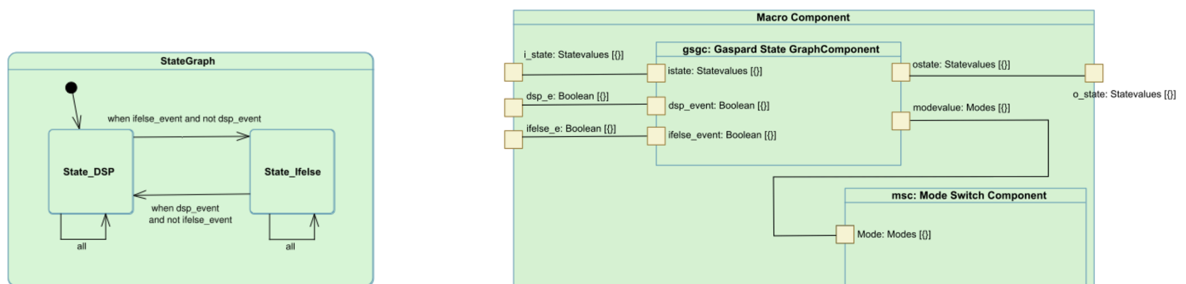


FIGURE 4.3 – Définition d'un Mode Switch Component (MSC) dans Gaspard, montrant une liaison de type "un mode = une implémentation"



(a) Définition d'un Gaspard State Graph (GSG)

(b) Vue en composant montrant la connexion entre les GSG et MSC définis précédemment

FIGURE 4.4 – Modélisation d'un composant reconfigurable dans Gaspard

À noter que le pattern *Strategy* est également mappable – en étant moins synthétique cependant – sur une machine à état, dont tous les états sont connectés entre eux et permettent de transiter vers tout état sur occurrence de son identifiant.

L'approche GASPARD adopte de son côté une approche non-standard pour traiter ce type de reconfiguration. Toute tâche reconfigurable encapsule deux composants GASPARD : un composant nommé *Mode Switch Component* (MSC) auquel sont rattachées des implémentations via des collaborations UML, cf. figure 4.3, et un composant nommé *Gaspard State Graph* (GSG) (assimilable à un MARTE *ModeBehavior*) tenant rôle de machine à état (cf. figures 4.4a et 4.4b). MSC et GSG communiquent ensemble via leurs ports typé *Mode* : sur occurrence d'événements, le GSG adapte son comportement et transmet l'identifiant de son mode actif au MSC qui sélectionne l'implémentation liée à ce mode via une collaboration UML.

Le ModeBehavior est le composant de type machine à état dans MARTE. Sa sémantique est quasi-identique à celle des UML States Machines. Il est à noter que ces deux approches n'exploitent pas la notion de *mode* – telle qu'elle est définie dans le profil MARTE – à son plein potentiel. Un *mode* est une définition explicite d'un aspect de configuration du système : sous-composants (par exemple des implémentations de tâches) et allocations vers des ressources matérielles, changements de valeurs de propriétés, ajout/suppression de connections, etc.

En effet pour ce type de reconfiguration, GASPARD et MoPCoM se restreignent à définir des ModeBehaviors – respectivement des UML States Machines – liés à une seule tâche. La composition synchrone de comportements ne peut alors être représentée explicitement ; par exemple, un mode (ou état) ne peut représenter l'affectation d'implémentations pour plusieurs tâches à la fois. Or cela devrait être possible étant donné la sémantique de mode dans MARTE.

En raison de leur proximité sémantique avec les automates de mode (qui sont à terme, le langage visé pour la représentation formelle du comportement), les ModeBehaviors sont tout à fait appropriés pour la modélisation comportementale. Notre approche se propose d'aller plus loin dans l'exploitation des machines à états que dans MoPCoM et GASPARD en décorrélant les définitions structurelles et comportementales des tâches autorisant ainsi à représenter des aspects plus poussés de configurations que la simple association *un mode = une implémentation*.

Dans FAMOUS la modélisation de l'application consiste alors à représenter un graphe de tâches (RtUnits) échangeant des flots de données (via des FlowPorts), ainsi qu'un flot d'événements dédié au contrôle (via des ModeBehaviors dont chaque mode est lié à une définition d'une partie d'une configuration du système). L'exécution de ces deux flots s'effectue en séquence dans des instants logiques : à chaque instant logique, les événements de contrôle sont consommés par les ModeBehavior qui calculent leur nouveau mode actif ; la composition des modes actifs donne la configuration globale du système à affecter avant d'exécuter la séquence des tâches traitant le flot de données ; le traitement du flot de données est piloté par le mécanisme de diffusion de configuration qui propage l'ordre de reconfiguration en respectant l'ordonnancement imposé par le modèle de calcul. Dans le cadre d'un modèle de tâche en réseau de Kahn, toute tâche de l'application réagit selon le principe exposé en figure 4.5 : l'exécution de l'application s'effectue en boucle infinie, chaque itération de cette boucle déclenchant une séquence complète du graphe de tâches ; en début de boucle, si des événements de contrôle sont reçus depuis l'environnement alors ils sont traités par le contrôleur de reconfiguration – synchronisant les ModeBehaviors – qui produit un ordre de reconfiguration correspondant. Si la configuration demandée est différente de la configuration courante, elle doit être propagée afin que toute tâche ne traite

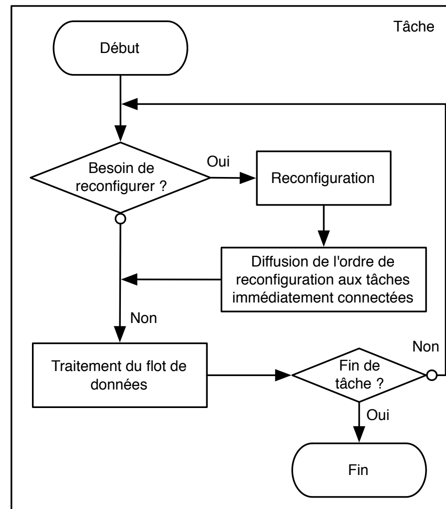


FIGURE 4.5 – Comportement d'une tâche générique, acceptant (s'il y a lieu) un ordre de reconfiguration et le diffusant à ses tâches immédiatement connectées dans le graphe

de nouvelle données qu'après reconfiguration ; l'ordre est alors propagé aux tâches sources du réseau de Kahn, qui se reconfigurent, puis traitent leurs entrées (flot de données) et distribuent l'ordre de reconfiguration aux tâches qui leur sont immédiatement connectées. La reconfiguration s'effectue selon le même principe pour les tâches suivantes jusqu'à ce que tout le graphe soit traité, ce qui se produit obligatoirement étant donné que toute tâche source est notifiée, et que le graphe n'est pas sensé contenir de cycle. La synchronisation des ModeBehaviors pour le contrôle de leur composition sera vue plus loin dans ce chapitre.

Définition des configurations partielles

La liaison des modes à une définition de configuration partielle est établie via un élément MARTE nommé *Configuration*.

Une *Configuration* MARTE est représenté par une structure composite UML stéréotypée par «Configuration». Ce stéréotype ajoute l'attribut *mode* à la structure et permet donc de la lier (par valuation de cet attribut) à un mode d'un ModeBehavior donné.

À l'intérieur d'une telle structure, il est possible de définir graphiquement des sous-composants et leurs connexions, des flots, ou des valeurs de propriétés. Au niveau application, l'intérêt est ici de pouvoir définir l'implémentation effective d'une (ou plusieurs) tâche(s) dans un mode donné. À titre d'exemple, considérons trois tâches, T1, T2 et T3. la figure 4.6 montre quatre Configurations MARTE, $C1$, $C2$, $C3$ et $C4$, respectivement liées aux modes $M1$, $M2$ et $M3$, $M4$ de deux ModeBehaviors ; La figure 4.7 donne la vue en composant des Configurations MARTE, spécifiant les aspects concrets de configurations : en mode $M1$, la Configuration MARTE $C1$ à pour sémantique d'imposer respectivement

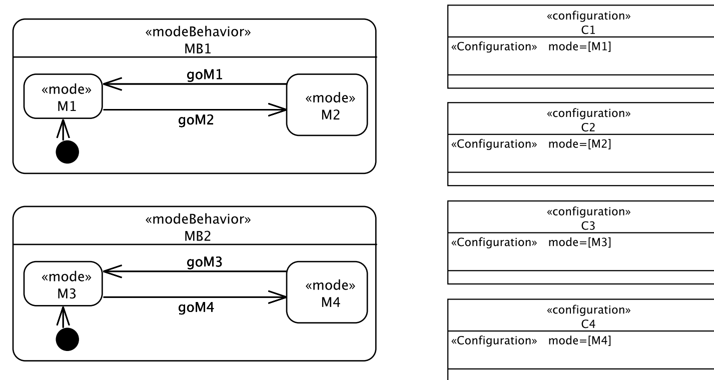


FIGURE 4.6 – Définition de ModeBehaviors et de Configurations MARTE associées

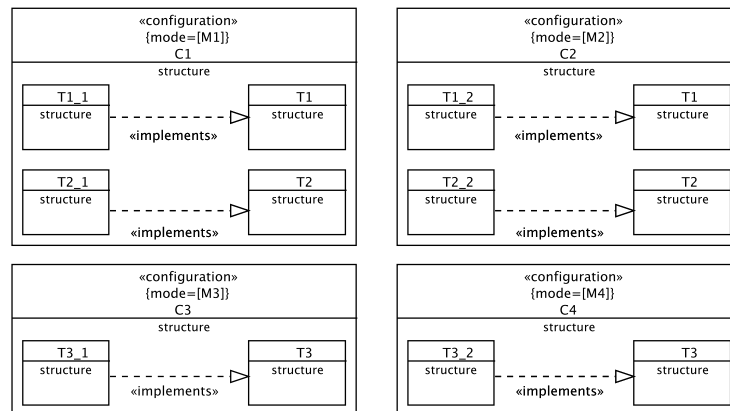


FIGURE 4.7 – Vue en composant des Configurations MARTE, définissant les aspects de configurations à mettre en œuvre lorsque leur mode associé est actif

aux tâches $T1$ et $T2$ les implémentations $T1_1$ et $T2_1$ (ces implémentations peuvent par exemple être définies par des classes UML respectivement encapsulées et connectées aux ports de $T1$ et $T2$, dans une approche similaire à MoPCoM) . Conformément à cette sémantique, en $M2$, les tâches $T1$ et $T2$ sont instanciées par $T1_2$ et $T2_2$. $T3_1$ est l'implémentation désignée pour la tâche $T3$ en mode $M3$, $T3_2$ en mode $M4$.

Une Configuration MARTE est dite *partielle*, dans le sens où elle peut se limiter à définir un aspect particulier de la configuration du système. Pour un instant logique donné, la configuration effective du système est donc celle donnée par les Configurations MARTE associées aux modes actifs de tous les ModeBehaviors définis dans le modèle.

Les configurations effectives sont mises en œuvre lors de la diffusion d'un ordre de reconfiguration et ne sont donc pas traitées ici, la diffusion étant supposée correcte. Seule une abstraction logique de la notion de *configuration du système* est retenue dans cette étude : la combinaison de ses modes actifs à un instant donné. Si de telles combinaisons

peuvent être établies en amont (c'est-à-dire avant chaque séquence d'exécution de tâches) alors elles peuvent être communiquées au dispositif de diffusion de configurations qui possède alors, via les liens établis avec les Configurations MARTE, tous les éléments pour effectuer sa mission.

4.2.2 Modélisation de l'architecture

À ce stade d'évolution du projet FAMOUS, de nombreux points nécessitent encore d'être formalisés quant aux fondations retenues pour la modélisation d'architectures reconfigurables dans MARTE. De nouvelles sémantiques et extensions nécessitent en effet d'être définies afin de pallier aux manques du profil dans ce domaine. Mais dans le cadre de cette étude, ceci ne doit pas être impactant : le modèle d'architecture est ici vu comme une collection de composants (processeurs, co-processeurs, bus, etc.) connectés entre eux via des ports. Une certitude étant que certains composants d'architecture auront la possibilité d'être identifiés comme *reconfigurables*, ce qui aura une importance sémantique dans le cadre de l'allocation d'un composant applicatif vers un composant d'architecture.

Dans les sémantiques qui seront proposées à terme, si des éléments architecturaux d'un modèle disposent de capacités à être modifiés en ligne, alors ces comportements devront être spécifiables au moyen de machines à état, donc de ModeBehaviors dont les modes sont connectés à des Configurations MARTE, lesquelles représentent les diverses formes adoptables par les composants d'architecture. Typiquement, on peut envisager de représenter ainsi des composants dont la taille mémoire peut évoluer, ou dont le débit de communication avec d'autres composant peut être adapté.

Les modes des ModeBehaviors définis dans le PM à cet effet se combinent naturellement avec ceux dont les ModeBehaviors appartiennent au PIM, afin de former une configuration du système.

4.2.3 Modélisation de l'allocation

L'allocation dans MARTE désigne la connexion d'éléments applicatifs vers la plateforme. Cela consiste en un ensemble de dépendances UML, annotées par le stéréotype MARTE «allocate». Le PSM, troisième et dernier sous-modèle d'un modèle MARTE, est le regroupement de l'ensemble des informations d'allocations entre le PIM et le PM. L'objectif du PSM est de pouvoir générer l'ensemble du système, c'est-à-dire la plateforme et le logiciel capable d'y être exécuté.

La figure 4.8 montre l'allocation de la tâche $T1$ (issue du PIM) sur un processeur Xilinx Microblaze (défini en tant que composant stéréotypé par «HwProcessor» dans le PM). Si une tâche est reconfigurable, c'est-à-dire si elle dispose de plusieurs implémentations pos-

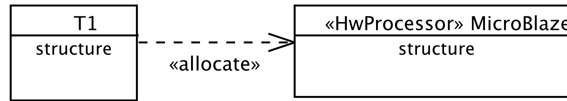


FIGURE 4.8 – Modélisation de l'allocation de la tâche T1 sur un processeur MicroBlaze (élément de modèle devant être présent dans le PM)

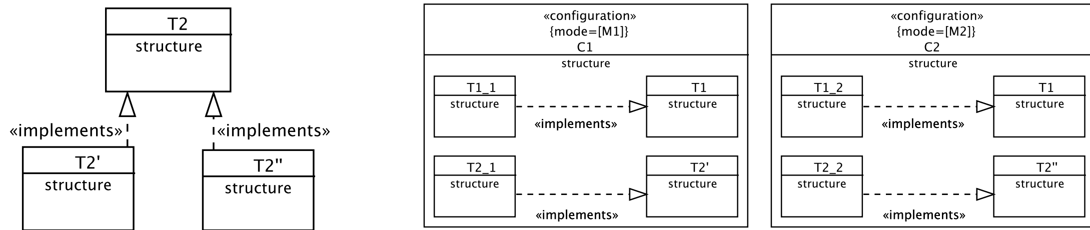


FIGURE 4.9 – Définition d'une hiérarchie de tâches implémentant T2, et adaptation des Configurations MARTE pour refléter leur mise en œuvre concrète suivant l'activation de leur mode respectif

sibles, l'allocation d'une telle tâche à une ressource donnée impose à ses implémentations d'être compatibles avec cette ressource. Pour $T1$, le Microblaze étant un processeur de type GPP, cela signifie que $T1_1$ et $T1_2$ doivent être des implémentations logicielles. Si les implémentations d'une tâche ne sont pas toutes compatibles avec la ou les mêmes ressource(s), il est alors nécessaire de construire une hiérarchie de sous-tâches liées directement ou indirectement à cette tâche par une connexion stéréotypée «implements» ; la sémantique signifiant que toute sous-tâche dispose des ports de la (ou des) tâche(s) ciblée(s) par ce lien.

À titre d'exemple, la tâche $T2$ dispose d'une implémentation logicielle, $T2_1$, et d'une matérielle, $T2_2$. $T2$ ne peut donc pas être allouée au Microblaze, pourtant nécessaire à l'exécution de $T2_1$, car $T2_2$ est incompatible avec cette ressource. Il est donc nécessaire d'apporter une modification du PIM pour $T2$ afin de représenter une hiérarchie comportant deux sous-tâches $T2'$ et $T2''$, auxquelles sont associées respectivement les implémentations $T2_1$ et $T2_2$, tel que montré en figure 4.9. Dès lors, il est autorisé d'allouer $T2'$ et $T2''$ respectivement à un MicroBlaze et à un co-processeur [28] Co-pro (stéréotypé en MARTE par «HwPLD»), cf. figure 4.10.

À ce niveau de la modélisation, le concepteur à la possibilité de définir des Configurations MARTE encapsulant des allocations. Les éventuels ModeBehaviors donnés par le concepteur à ce niveau permet de spécifier des comportements de relocation (changement

28. Qui devra être identifié comme reconfigurable dans la modélisation proposée par FAMOUS.

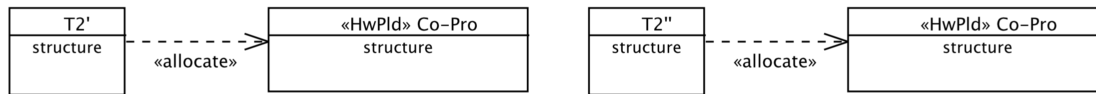


FIGURE 4.10 – Allocation de chaque nouvelle tâche définie dans la hiérarchie vers leur élément de plateforme respectif

de l'allocation d'un élément du PIM vers un autre élément du PM), ce qui est propre au PIM. Au final, pour un instant logique donné, la combinaison des modes actifs des ModeBehaviors du PIM, du PM et du PSM forme la configuration logique du système dans cet instant.

4.2.4 Ajout de propriétés et contraintes

La sélection sûre d'une configuration nécessite de connaître les règles et contraintes comportementales devant être respectées à l'exécution. Ces contraintes peuvent être explicites – opérant directement sur la notion de mode actif/inactif (séquences d'activation, exclusions mutuelles, etc.) – ou implicites – requérant des informations non fonctionnelles (critères de qualité de service, de consommation d'énergie, etc.) sur les configurations du système afin de spécifier des objectifs quantitatifs à respecter –.

Le profil MARTE dispose d'un moyen de définir ces propriétés et les associer à des composants UML : les Non-Functional Properties (NFP). Les NFP sont valuées par le concepteur directement dans ses modèles au moyen d'un langage interne à MARTE : Value Specification Language (VSL).

L'élément MARTE "Configuration" est naturellement approprié pour encapsuler les NFP afin d'associer des valeurs à des aspects d'une configuration. Tous les stades de granularité sont supportés par les Configurations MARTE : des plus fins, où l'on ne représente qu'un aspect particulier (allocation, sélection d'implémentation de tâche, etc.) à combiner avec d'autres pour former une configuration, jusqu'aux plus élevés, où une Configuration MARTE peut définir une configuration complète du système.

Non-Functional Properties

Dans MARTE, le modèle complet d'un système est décrit au moyen d'éléments de modèle (ressources, comportements, opérations, modes, etc.) et de propriétés sur ces éléments. Ces propriétés sont généralement distinguées en deux catégories : les *propriétés fonctionnelles*, qui concernent l'objectif de l'application (ce qu'elle effectue à l'exécution),

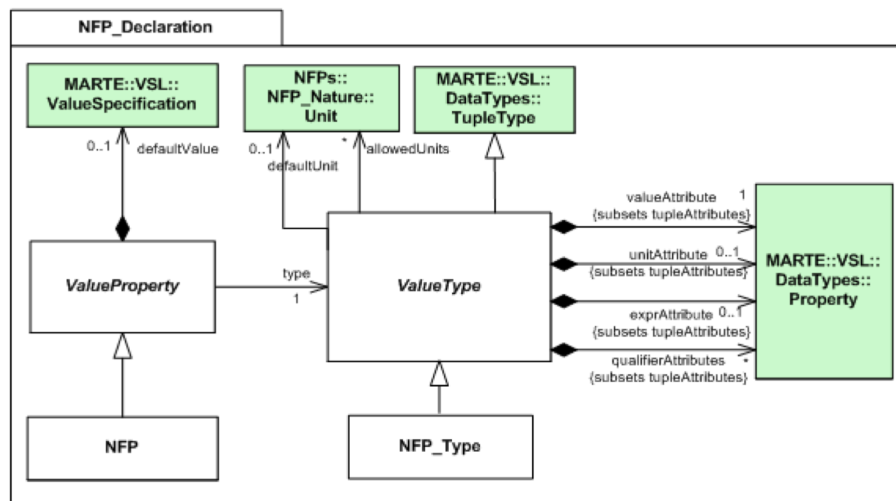


FIGURE 4.11 – Package MARTE NFP_Declaration, dédié à la définition de types et valeurs de propriétés non fonctionnelles

et les *propriétés non-fonctionnelles* (NFP), caractérisant les aptitudes des éléments associés à répondre à un objectif (la qualité d'exécution).

Les NFP permettent ainsi de préciser différentes caractéristiques d'exécution d'un ensemble d'éléments, telles que la bande passante, les délais d'exécution, les deadlines, l'utilisation de mémoire, la consommation d'énergie, etc.

MARTE dispose d'un package, NFP_Declaration (cf. figure 4.11), définissant la sémantique d'attribution d'un type et de valeurs à une NFP. On y distingue notamment le stéréotype NFP_Type, applicable sur toute définition de type de donnée (DataType) UML. Au sein du framework NFP, une librairie de types de base est définie en standard, la figure 4.12 en montre un extrait. Tout type NFP hérite de NFP_CommonType, et dispose des caractéristiques du stéréotype UML «dataType» : toute instance de ces types est donc associable à un élément de modèle en tant que propriété. Le stéréotype MARTE «nfpType» ajoute (entre autre) la capacité de spécifier l'attribut réservé à la valuation d'une NFP : à la déclaration d'un type NFP, il suffit de valuer l'attribut *valueAttrib* par le nom de l'attribut dédié au stockage de l'éventuelle valuation donnée par le concepteur.

Il existe deux manières d'attribuer une NFP à un élément de modèle : par ajout de propriété stéréotypée «nfp» dans le champs de propriétés du composant ciblé, ou par *annotation* au moyen d'un commentaire UML stéréotypé «nfpConstraint». La méthode par annotation permet en outre de préciser si la propriété définie doit être respectée par le comportement de l'élément ciblé (en valuant l'attribut *kind* du stéréotype par "contract"), ou respectée lors de l'utilisation de l'élément (en valuant l'attribut *kind* par "required").

la figure 4.13 montre un exemple d'annotation d'un «RtUnit» : *MyNfpConstraint1*

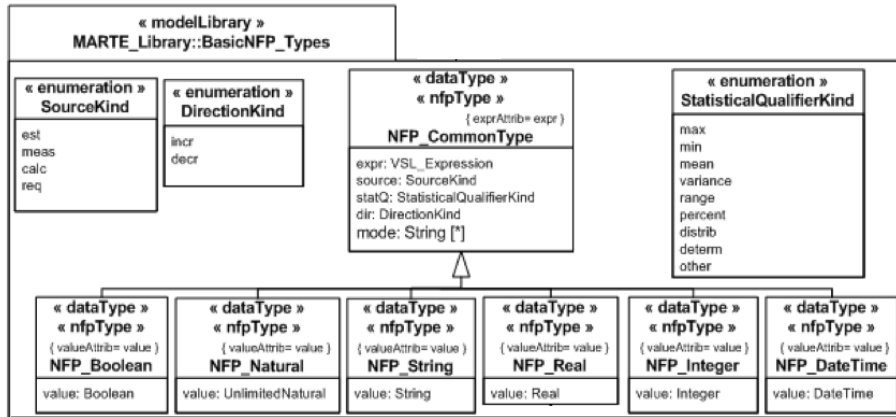


FIGURE 4.12 – Extrait de la librairie de types de NFP de base dans MARTE

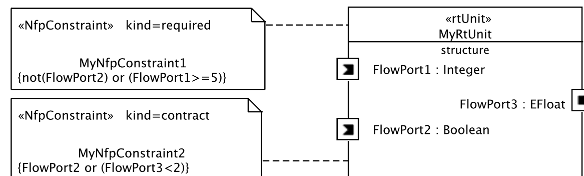


FIGURE 4.13 – Définition de contraintes temporelles à un composant ; "contract" signifie que la contrainte est respectée par le composant lui-même, "required" signifie que la contrainte est supposée être respectée par l'environnement du composant

indique que l'utilisation de *MyRtUnit* dans un graphe de tâches suppose que pour tout instant logique, si la valeur de *Flowport2* donnée en entrée est vraie, alors celle donnée à *FlowPort1* doit être supérieure ou égale à 5 ; *MyNfpConstraint2* spécifie quant à elle que *MyRtUnit* s'engage à produire une valeur inférieure à 2 sur son port de sortie *FlowPort3* lorsque *FlowPort2* est valué à faux. la figure 4.14 montre un exemple de composant, *MyComponent*, dans lequel sont associées deux propriétés, *myNfp1* et *myNfp2*, stéréotypées «nfp», *myNfp1* étant de type basique NFP_Integer et valuée à 10, *myNfp2* étant de type NFP_Duration dont les attributs *value*, *unit*, et *precision* sont respectivement valués à 2.1, s, et 0.01. La syntaxe de valuation des attributs d'une NFP consiste ici à rappeler chacun d'eux entre accolades, puis les faire suivre du symbole = et d'une valeur sous forme d'une chaîne de caractères. Qu'il s'agisse d'une annotation ou d'une propriété, la syntaxe de valuation d'une NFP est celle de VSL.

Value Specification Language

Le langage d'expression VSL défini dans MARTE adresse les points suivants :

– spécification de paramètres, constantes, et expressions sous forme textuelle ;

MyComponent
«nfp» myNfp1 : NFP_Integer = {value=10} «nfp» myNfp2 : NFP_Duration = {value=2.1, unit=s, precision=0.01}

FIGURE 4.14 – Association de valeurs de propriétés non fonctionnelles à un composant

- relations entre paramètres ou constantes, définies au moyen d'expressions arithmétiques, logiques, relationnelles, ou conditionnelles ;
- définition d'assertions temporelles ;
- spécification de valeurs composites (collections, intervalles, tuples).

La syntaxe de VSL est définie dans le profil MARTE [49]. Il s'agit ici d'employer un sous-ensemble de ce langage (la syntaxe employée jusqu'ici) afin de rendre possible la transformation de spécifications VSL vers une représentation synchrone.

La spécification de valeurs, composites ou non, servira à la définition de *poids* (valuation de propriétés non-fonctionnelles associées à des Configurations MARTE). Les assertions temporelles telles que représentées dans les NfpConstraints de la figure 4.13 – expressions logiques reposant sur des valeurs d'événements, l'état du système, des propriétés non fonctionnelles de configurations, etc. – seront employées pour la définition de contraintes internes ou externes du système.

4.3 Extension du profil MARTE

Les éléments standards MARTE venant d'être présentés permettent de modéliser le comportement et les contraintes comportementales du système. Cependant, il manque deux points importants avant de pouvoir transformer une telle spécification en un système reconfigurable exécutable. Le premier point concerne la combinaison des NFP définies sur les Configurations MARTE : ces éléments du modèle peuvent tout à fait caractériser un aspect particulier d'une configuration complète, il est alors indispensable de pouvoir combiner les NFP des Configurations MARTE dont les modes associés sont actifs afin de calculer ceux de la configuration complète, dans le but de vérifier les éventuelles contraintes comportementales impactées. Une telle sémantique est absente de MARTE, et n'a été proposée ni dans MoPCoM ni dans Gaspard, il est nécessaire ici d'augmenter le profil afin de permettre au concepteur de définir un moyen de combinaison automatique de NFP.

Le second point concerne l'absence de sémantique de contrôle. Bien que les divers aspects comportementaux du systèmes soient définis à l'aide de ModeBehaviors, les éventuelles synchronisations ne sont pas représentées et le flot de réception d'événements de contrôle est absent. Dans MoPCoM comme dans Gaspard, le contrôle de la reconfigu-

ration est piloté par un RtUnit approprié, mais dans les deux cas, les flots de contrôle et de données sont liés (synchronisés sur les entrées des tâches) ce qui force les deux flots à suivre le même modèle de calcul. Dans l'approche FAMOUS, il est proposé de séparer les notions de "contrôle" et de "diffusion" de reconfiguration. Il doit notamment être permis d'exécuter le contrôle en parallèle des tâches traitant purement le flot de données, la synchronisation des reconfigurations s'effectuant sous la responsabilité du système de diffusion.

Cette section s'attache à présenter les améliorations ou les moyens d'utilisation du profil MARTE pour traiter les combinaisons de poids et la sémantique de contrôle de la reconfiguration.

4.3.1 Composition générique des propriétés non fonctionnelles

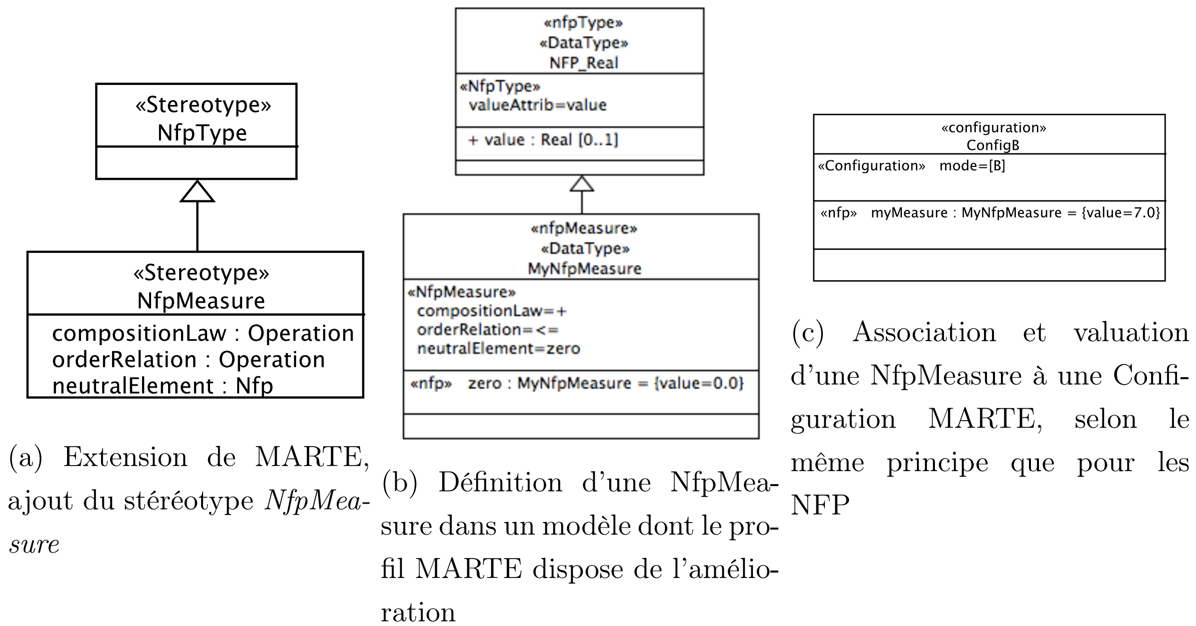
Une solution de modélisation n'impliquant aucune modification du profil MARTE pour traiter la composition consiste à reporter cette responsabilité du côté du concepteur : celui-ci doit définir toutes configurations complètes possibles – ou se restreindre à un sous-ensemble de configurations – dans des Configurations MARTE pour lesquelles il établit les valeurs des propriétés non fonctionnelles qu'il souhaite considérer. Ce choix d'exposer les configurations du système dans le modèle a notamment été retenu dans GASPARD.

Le problème d'un tel choix provient du fait que plus le système à modéliser contient d'éléments reconfigurables à synchroniser, plus le nombre de configurations complètes est conséquent. Ici, ce nombre croît de manière exponentielle par rapport au nombre de ModeBehaviors. Pour de tels systèmes de grande taille, le concepteur est incité à ne considérer qu'un sous-ensemble de configurations complètes possibles, ce qui revient à sous-optimiser les potentialités de reconfigurations.

Si l'approche modulaire présentée ici – dans laquelle des aspects de configurations peuvent être définis en vue d'être combinés – pouvait bénéficier d'une sémantique de composition sur les NFP, le concepteur pourrait s'abstenir de réaliser manuellement leurs combinaisons.

Pour ce faire, il est nécessaire d'augmenter le profil MARTE afin de permettre la spécification d'informations complémentaires pour la combinaison automatique. La figure 4.15a montre ainsi la proposition d'extension du stéréotype NfpType vers une nouvelle notion nommée NfpMeasure. Une NfpMeasure est un type de NFP particulier il s'agit d'une structure caractérisant un ensemble (désigné par le type de NFP) muni d'une loi de composition interne (l'opération *compositionLaw*), d'une relation d'ordre (l'opération *orderRelation*) et d'un élément neutre (la NFP *neutralElement*).

Avec une telle extension, le concepteur peut alors définir ses propres NfpMeasures, de la même manière que pour ses propres NFP. La figure 4.15b montre un exemple de dé-



(a) Extension de MARTE, ajout du stéréotype *NfpMeasure*

(b) Définition d'une NfpMeasure dans un modèle dont le profil MARTE dispose de l'amélioration

(c) Association et valuation d'une NfpMeasure à une Configuration MARTE, selon le même principe que pour les NFP

FIGURE 4.15 – Amélioration du profil MARTE pour le support des combinaisons de valeurs de NFP

finition d'une NfpMeasure *MyNfpMeasure*, étendant les propriétés de NFPReal (élément existant dans la librairie de types de NFP basiques), pour laquelle le concepteur choisit d'employer l'addition (définie dans NFPReal) pour loi de composition interne, l'opération <= (également définie dans NFPReal) pour relation d'ordre, et l'élément neutre *zero* ciblant une instance de MyNfpMeasure (soit une NFP), valuée par "0.0" (soit une valeur acceptable pour NFPReal), et encapsulée dans MyNfpMeasure. La figure 4.15c montre un exemple d'affectation d'une NfpMeasure à une configuration. On remarque que la syntaxe est identique à celle d'une valuation de NFP. L'extension proposée a donc pour autre avantage le fait de ne pas perturber l'éventuel outillage de valuation ou d'analyse de NFP employé par le concepteur si celui-ci choisit de définir des NfpMeasures pour caractériser ses configurations, car tout instance de NfpMeasure est assimilable à une NFP.

Cette sémantique proposée va non seulement permettre de créer un outillage adapté au calcul automatique des combinaisons de NFP pour les Configurations MARTE en utilisant les opérations de composition et éléments neutres, mais également d'exploiter les relations d'ordres afin d'optimiser les choix de reconfiguration.

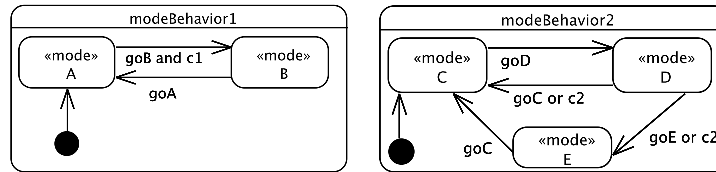


FIGURE 4.16 – Définition de deux ModeBehaviors à synchroniser

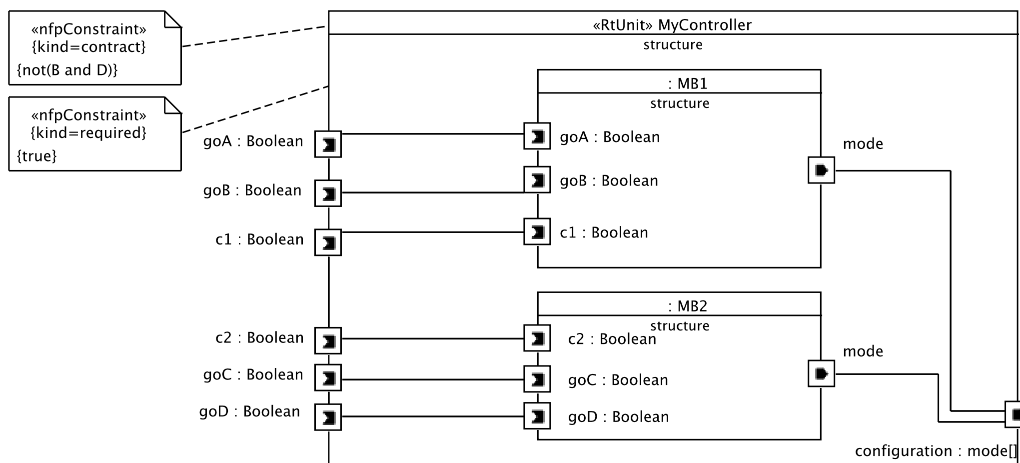


FIGURE 4.17 – Synchronisation à l'aide d'un RtUnit et spécification de contraintes d'exécution

4.3.2 Modélisation d'un contrôleur générique

Afin de rendre le comportement du système exécutable, il est nécessaire de définir le flot de contrôle, c'est-à-dire les émissions/réceptions d'événements traités par les éléments de modélisation dédiés au comportement (ici, les ModeBehaviors).

Le premier besoin d'organisation du contrôle comportemental est celui de la synchronisation : le comportement du système peut être défini de manière modulaire à l'aide de multiples ModeBehaviors, et ceux dont les réactions sont supposées s'effectuer dans le même instant logique doivent disposer d'un flot d'événements synchronisés.

Cette sémantique de synchronisation étant supportée par les RtUnits, l'emploi d'un RtUnit pour encapsuler les interfaces de ModeBehaviors synchronisés est appropriée. La figure 4.16 donne un exemple de deux ModeBehaviors, dont la synchronisation (vue composite) est représentée au sein d'un RtUnit, nommé *MyController*, en figure 4.17. La sémantique d'exécution suit donc celle définie par les propriétés du stéréotype «RtUnit», ici par défaut à chaque exécution ce RtUnit reçoit un et un seul événement par port d'entrée et produit une configuration en sortie, soit la combinaison des modes actifs encapsulés.

À partir d'une telle représentation, les RtUnits ainsi dédiés à la synchronisation de ModeBehaviors sont les éléments idéaux pour y associer des éventuelles contraintes d'exécution. Cette même figure 4.17 montre un exemple d'association de deux NfpConstraints, l'une (facultative) précisant qu'aucune contrainte particulière n'est imposée sur l'utilisation de ce RtUnit (puisque sa contrainte temporelle définie est toujours vraie), et l'autre indiquant que ce RtUnit s'engage à ne jamais rendre actifs les états B et D au même instant.

De telles contraintes ne sont ici que pures spécifications. En effet, depuis la configuration initiale $\{A;C\}$, rien n'empêche a priori les événements goA, $c1$ et goD d'être vrais au même instant, ce qui amènerait à activer B et D à la fois, or cela est interdit. Il est donc nécessaire, afin d'assurer la contrainte temporelle $not(BandD)$, de piloter les événements en entrées de ce RtUnit. Ce pilotage peut être effectué de manière externe, dans ce cas il est nécessaire d'adapter la contrainte temporelle $required$ afin de refléter une hypothèse correcte d'utilisation. Il peut également être interne, les événements requis par les ModeBehaviors synchronisés proviennent alors soit de l'interface de leur RtUnit (donc de l'extérieur), soit depuis un composant interne, ayant la visibilité sur l'interface du RtUnit et pouvant disposer de sa propre interface, dont le rôle est de calculer des valeurs correctes d'événements afin de les fournir aux ModeBehaviors.

La figure 4.18 montre un tel exemple de composant interne, appelé *resolver*. Celui-ci, au même titre que les ModeBehaviors encapsulés, s'exécute à la même cadence que *MyController* : à chaque exécution de MyController, les événements goA, goB, goC, goD et $otherInput$ sont reçus depuis l'environnement ($otherInput$ est ici à titre d'exemple pour montrer qu'il est possible de dédier une interface au composant interne), puis *resolver* calcule les valeurs appropriées pour $c1$ et $c2$. Ainsi, tous les événements requis par les deux ModeBehaviors sont prêts pour être consommés au même instant logique.

À ce point de la spécification, il est nécessaire pour le concepteur d'implémenter le comportement du *resolver* (l'aide de ModeBehaviors ou bien en implémentant ce composant dans un langage particulier après transformation de cette spécification vers du code).

Cette approche d'implémentation manuelle est typique dans la communauté de la vérification formelle : le comportement complet du système muni de contraintes et d'un contrôle interne sont spécifiés, puis vérifiés (notamment via des outils de model-checking). Cette méthode pose un problème principal : la réalisation manuelle du contrôle interne (*resolver*) est susceptible d'être sous-optimale. Deux raisons à cela :

– l'espace des configurations est trop large et complexe à appréhender, le concepteur est alors obligé de se restreindre à contrôler un sous-espace ;

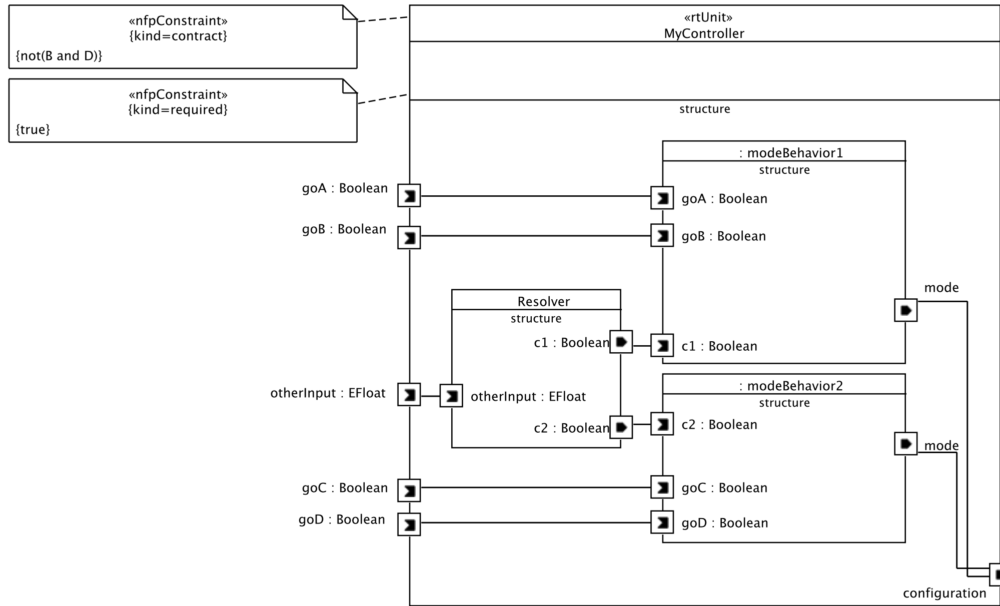


FIGURE 4.18 – Définition d'un élément interne au RtUnit de contrôle, dédié à la résolution des valeurs d'événements internes

– l'erreur est humaine, même sur un exemple "petit", le concepteur peut oublier d'autoriser l'accès à des configurations légales.

Concernant la deuxième raison, si l'on prend pour exemple l'implémentation suivante pour le *resolver* : $c1 = false$ [29], c'est-à-dire que l'on attribue la valeur *false* à $c1$ à chaque instant logique et ce quelles que soient les valeurs des autres événements, alors une telle implémentation sera vérifiée formellement comme étant valide. En effet, depuis l'état initial $\{A;C\}$ il sera impossible de rejoindre $\{B;D\}$. Cependant, $\{B;C\}$ et $\{B;E\}$ sont inaccessibles alors qu'une valuation appropriée de c1 à *false* seulement quand cela est nécessaire aurait pu permettre l'accès à ces configurations directement ou indirectement depuis la configuration initiale. Pour remédier à ce problème, l'emploi d'outils de synthèse de contrôleur est nécessaire : il s'agit d'obtenir automatiquement le code de *resolver*, un code à la fois correct et *maximalement permissif*, dans le sens où les valeurs produites par le *resolver* ne sont forcées à *true* ou *false* que lorsque cela est nécessaire.

Dans cette approche, il n'est donc plus obligatoire pour le concepteur d'implémenter le *resolver*. Une modification légère du profil MARTE est toutefois nécessaire afin de permettre un tel outillage : les RtUnits dédiés au contrôle de la reconfiguration doivent être distingués des RtUnits applicatifs, les composants internes dédiés au calcul d'événements internes doivent être identifiables, et enfin les types de NfpMeasures requis pour les règles

29. On ne s'occupe pas de c2 dans cet exemple. Quelle que soit la valeur attribuée à c2, le remarque est la même.

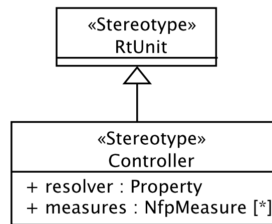


FIGURE 4.19 – Extension de MARTE pour compléter le support en terme de spécification de contrôle

comportementales d'un RtUnit doivent être déclarés dans ce RtUnit. Ce dernier point permet notamment de ne calculer que les combinaisons de NfpMeasures nécessaires : par exemple, si deux types de NfpMeasures, admettons *power* et *qos*, sont définis pour chaque Configuration MARTE et qu'un RtUnit de contrôle dispose de règles comportementales (NfpConstraint) indépendantes de *power*, alors seules les valeurs de *qos* sont à calculer pour les combinaisons possibles de Configurations MARTE (liées à des modes) dans ce RtUnit.

La figure 4.19 montre l'extension proposée pour traiter un tel besoin. Le stéréotype «Controller» pourra être associé à tout RtUnit dédié au contrôle, permettant ainsi de le distinguer des autres. Ainsi identifiés, ces RtUnits pourront être exécutés indépendamment du modèle de calcul choisi pour l'exécution des tâches (autre RtUnits) de l'application. Les RtUnits de contrôle peuvent alors réagir simplement sur occurrence d'événements en provenance de l'environnement. Dans le flot d'exécution FAMOUS, la synchronisation des ordres de reconfigurations issus des ces RtUnits vers les tâches de l'application s'effectuera au moyen d'un mécanisme de diffusion capable de reconfigurer les tâches au bon moment pendant leur exécution. Dans cette étude, un tel mécanisme n'est pas encore effectif. C'est pourquoi dans les exemples d'exécution qui seront donnés par la suite, un mécanisme simple de diffusion sera mis en œuvre, conformément à l'approche présentée dans en fin de partie "Spécification comportementale" de la sous-section 4.2.1.

Tout RtUnit muni du stérétotype «Controller» pourra également identifier un composant interne (propriété UML) en tant que *resolver* – c'est à dire en tant qu'unité responsable de la valuation d'événements internes à ce RtUnit – et déclarer les types de NfPMeasures à calculer pour l'évaluation de ses contraintes. La figure 4.20 donne un exemple d'utilisation complet de ce stéréotype, basé sur l'exemple d'encapsulation des deux ModeBehaviors déjà présentés.

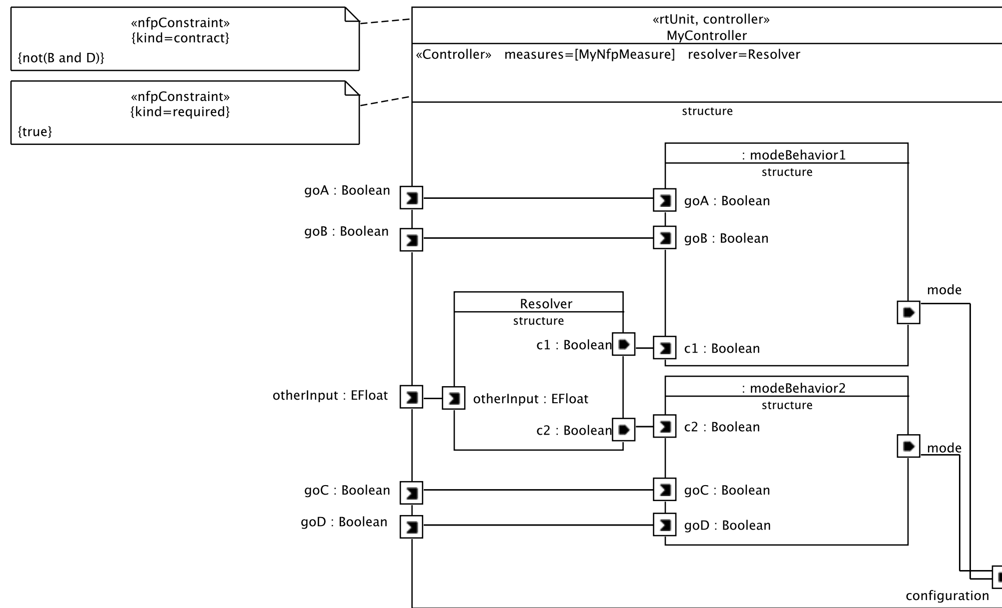


FIGURE 4.20 – Utilisation du nouveau stéréotype pour l'identification de ce RtUnit en tant qu'élément de contrôle, identification du resolver à synthétiser, et déclaration des types de NfpMeasures à combiner

4.4 Conclusion

Le présent chapitre a montré une méthodologie d'utilisation de MARTE pour la spécification du contrôle de reconfiguration d'éléments de modèles. Et ce, qu'ils proviennent du PIM, du PM ou du PSM. Cette méthodologie repose sur l'emploi de ModeBehaviors, dont chaque mode est une représentation logique d'un aspect de configuration, et de Configurations MARTE qui, associées à des modes, donnent la correspondance physique des aspects de configuration modélisés. Comparée aux approches existantes, la solution de modélisation proposée permet tout d'abord de séparer les notions de traitement de données par des tâches et le contrôle de leur reconfiguration. Les deux flots, contrôle et données, disposent chacun de leur propre modèle de calcul. Leur synchronisation s'effectuant au moyen d'un dispositif de diffusion de configuration, issu du flot de contrôle, vers les tâches traitant le flot de données. Le profil MARTE a également été étendu afin de supporter 1) une sémantique pour la combinaison de valeurs de propriétés non-fonctionnelles définies par le concepteur lors de la spécification de Configurations MARTE, et 2) une sémantique pour la spécification de contrôleurs de reconfiguration disposant d'éléments internes, nommés *resolver*, responsables du calcul d'événements internes. Si l'implémentation des *resolvers* peut être effectuée puis vérifiée manuellement par le concepteur, une approche à la fois automatique, correcte et optimale (*maximalement permissive*) peut cependant être réali-

sée à partir de la modélisation proposée. En effet, une telle spécification comportementale, associée à des contraintes (sur des NfpMeasures, sur des compositions de modes, etc.) et à des signaux externes (issus de l'environnement et donnés en entrée d'une tâche contrôle) et/ou internes (issus non pas de l'environnement mais seulement des tâches de contrôle) est un modèle typique sur lequel il est possible d'opérer une synthèse de contrôleur dans le but d'obtenir le code capable de valuer correctement les signaux internes, soit le code du *resolver*. C'est pourquoi la présente étude montrera par la suite comment outiller une telle sémantique pour synthétiser automatiquement le *resolver*.

5

Transformation du contrôle vers une représentation synchrone

Sommaire

5.1 Introduction **103**
5.2 Méta-modèle intermédiaire dédié au contrôle **104**
5.2.1 Automates de modes hiérarchiques 104
5.2.2 Combinaison des propriétés non fonctionnelles 106
5.2.3 Propriétés du contrôleur 108
5.3 Implémentation exécutable du contrôle **109**
5.3.1 Challenge de l'implémentation maximalement permissive . . 109
5.3.2 Transformation vers BZR et exploitation de la synthèse de contrôleur 114
5.4 Problème du non déterminisme de la représentation en automates **117**
5.4.1 Exemple 119
5.4.2 Différence sémantique entre les ModeBehavior et les automates en BZR 122
5.4.3 Exploitation du non-déterminisme pour la décision en ligne de reconfiguration 123
5.5 Conclusion **123**

5.1 Introduction

Si l'approche de conception dirigée par les modèles a pour premier objectif d'établir des spécifications de haut niveau d'abstraction d'un système donné, le deuxième objec-

tif – conformément à l'approche MDE – est de pouvoir transformer ces modèles vers d'autres représentations qui peuvent être intermédiaires (afin d'appliquer d'autres transformations ou bien d'employer un outillage d'analyse, de vérification ou de synthèse) ou finales (exécutables).

L'objectif de ce chapitre est de montrer comment un modèle reposant sur la sémantique MARTE introduite, munie de son extension, peut être transformé en une représentation intermédiaire afin de synthétiser les contrôleurs qu'il déclare. La sémantique MARTE étendue sera ici mentionnée par *RecoMARTE*. À noter qu'il s'agit ici d'une première version de RecoMARTE, spécialisée sur l'aspect *contrôle* ; d'autres contributions du projet FAMOUS viendront s'ajouter à cette proposition.

La première étape consiste à extraire du modèle RecoMARTE les informations de contrôle de reconfiguration, puis de les structurer en calculant ce qui doit l'être. À partir de cette structuration, la deuxième étape consiste à construire une représentation synchrone équivalente, au moyen d'un langage synchrone. Le langage utilisé ici est BZR, dont le compilateur a la particularité d'encapsuler un processus de synthèse de contrôleur.

Ces étapes clés ont notamment été définies pour la première fois dans [57], et concrétisées plus tard dans [55].

5.2 Méta-modèle intermédiaire dédié au contrôle

Cette section propose un métamodèle dédié à la préparation intermédiaire des informations de contrôle issues d'un modèle RecoMARTE. Il s'agit :

- d'extraire les ModeBehaviors
- d'établir les configurations logiques du système et de calculer leurs propriétés
- de construire, pour chaque RtUnit stéréotypé par «Controller», un modèle encapsulant ses propriété fondamentales (contraintes, ModeBehaviors pilotés, etc.)

L'idée est ici d'établir un modèle objet très proche de la cible, c'est-à-dire un modèle à partir duquel il sera simple d'effectuer une dernière transformation non pas "modèle vers modèle", mais "modèle vers texte" (modèle vers BZR).

5.2.1 Automates de modes hiérarchiques

L'extrait du métamodèle MARTE donné en figure 5.1 montre les aspects de modélisation comportementale liés aux ModeBehaviors. Plusieurs simplifications sont à opérer : 1) Les aspects de configuration définis par les Configurations MARTE sont abstraits dans la représentation synchrone car seules les configurations logiques (compositions de modes ac-

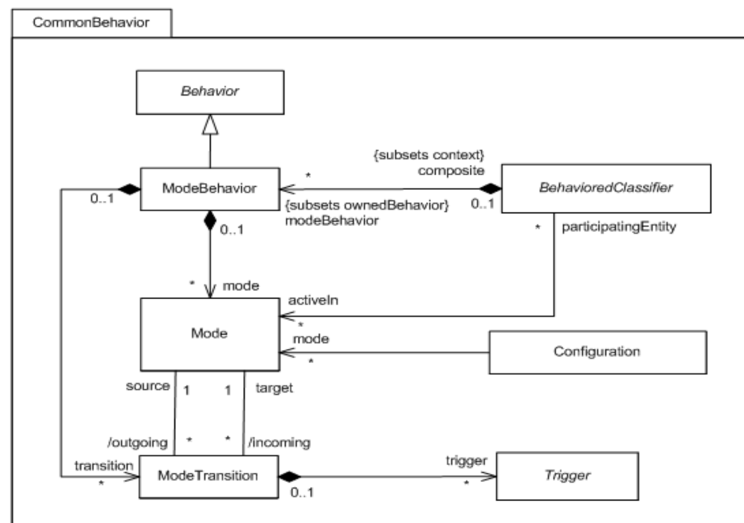


FIGURE 5.1 – Extrait du profil MARTE dédié à la spécification de comportement modal

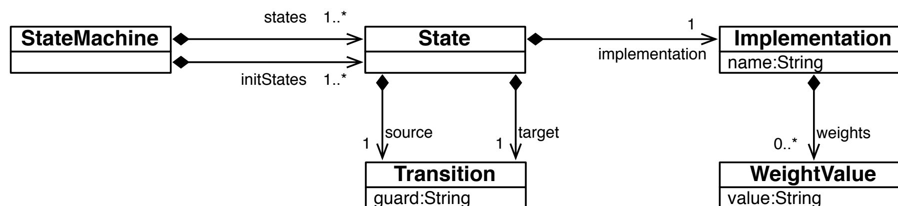


FIGURE 5.2 – Extrait du métamodèle intermédiaire pour la représentation comportementale

tifs dans un instant discret) sont traitées. 2) Les Triggers [30], c'est-à-dire les concepts liant des événements à l'activation d'un comportement (Mode) sont à préciser par le fait que la méthodologie proposée se restreint à les définir via des expressions booléennes exprimées en VSL. 3) Enfin, les ModeBehaviors héritant des propriétés des UML State Machines, ils peuvent déclarer plusieurs régions parallèles (chacune ayant son propre état initial) et connecter leurs modes à d'autres ModeBehaviors (un mode peut ainsi encapsuler un ModeBehavior) ; il faut donc tenir en compte cette sémantique de composition parallèle et hiérarchique pour la représenter dans le modèle objet intermédiaire, et établir les configurations (compositions de modes actifs) correspondantes. La figure 5.2 propose un extrait du métamodèle intermédiaire dédié à la représentation des machines à états.

La métaclasse *Implementation* correspond à une simplification de la Configuration MARTE, dans laquelle on encapsule uniquement un identifiant (*name*) et des éventuelles valeurs de propriétés non-fonctionnelles, appelée ici "poids" (élément *WeightVa-*

30. Le lecteur intéressé pourra se référer au package CommonBehavior dans MARTE.

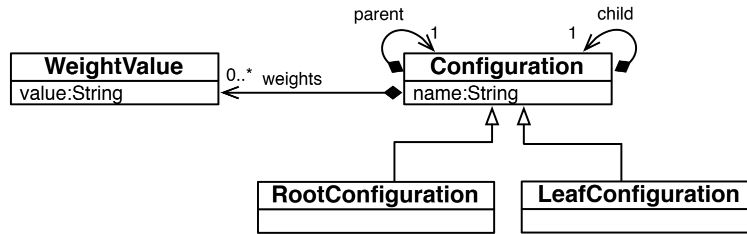


FIGURE 5.3 – Extrait du métamodèle intermédiaire pour la représentation des configurations

lue). Chaque implémentation est encapsulée par un élément *State*, comparable à la notion de Mode dont l'identifiant est le nom donné à l'implémentation.

La notion de *StateMachine* reprend celle du ModeBehavior, à la différence près qu'elle pointe directement sur un ensemble (non vide) d'états initiaux, faisant partie de sa collection d'états (*State*).

Enfin la métaclasse *Transition* désigne à la fois une fusion et une simplification des métaclasses *ModeTransition* et *Trigger* dans MARTE. Une transition dans ce métamodèle ne conserve que les notions de *source* et *target*, c'est-à-dire les états source et cible de la transition, ainsi qu'une garde en tant que propriété, désignant la condition de franchissement de cette garde si l'état source est actif. Cette garde est une simple chaine de caractères, désignant une expression booléenne exprimée en VSL.

5.2.2 Combinaison des propriétés non fonctionnelles

L'étape suivant l'extraction complète des ModeBehaviors et Configurations MARTE est de définir les configurations logiques du système – c'est-à-dire l'ensemble des ensembles possibles d'implémentations dont les modes peuvent être actifs dans un même instant discret – ainsi que leurs propriétés. L'extrait du métamodèle intermédiaire donné en figure 5.3 présente une métaclasse Configuration encapsulant un ensemble de valeurs de propriétés non-fonctionnelles et définissant une hiérarchie parent/enfant par des liens d'aggrégation (*parent* et *child*) vers d'autres éléments Configuration. Les deux métaclasses *RootConfiguration* et *LeafConfiguration* sont des configurations particulières, n'ayant pas d'existence concrète dans le système mais définissant les bornes respectivement supérieures et inférieures de tout configuration : il s'agit de de classer les configurations entre-elles à partir du calcul de leurs propriétés non-fonctionnelles (données par leurs implémentations sous-jacentes). Ces propriétés étant potentiellement de plusieurs types, le classement des configurations est alors multi-critères, et une structure adaptée pour un représenter un tel classement est un *treillis*, dont les bornes supérieures et inférieures sont donc représen-

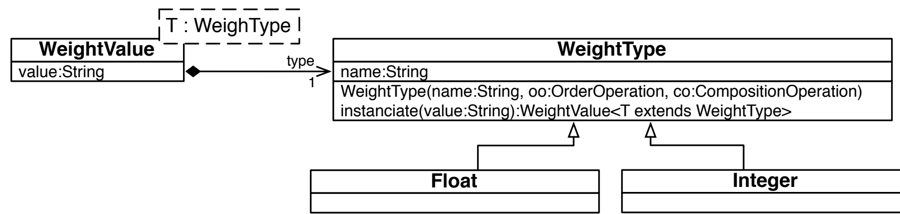


FIGURE 5.4 – Principe de définition de types dans le métamodèle intermédiaire

tées par *RootConfiguration* et *LeafConfiguration*. Un tel classement servira par la suite à optimiser les choix de configuration.

Si l'établissement des configurations possibles à partir des machines à état est un algorithme à la fois trivial et défini une fois pour toutes – il s'agit de calculer les combinaisons d'états possibles, une combinaison disposant d'un état par machine à état définie –, le calcul des propriétés (poids) de configurations à partir de celles de leurs implémentations associées puis le classement des configurations nécessite en revanche d'implémenter les opérations de d'ordre et de composition mentionnées dans le modèle RecoMARTE (cf. les opérations "loi de composition" et "opération d'ordre" de l'élément *NfpMeasure*). La figure 5.4 expose le principe de typage adopté dans le métamodèle intermédiaire, correspondant au typage de NfpMeasures et à leurs opérations. Toute valeur de poids (*WeightValue*) est associée à un type (*WeightType*) ; l'élément WeightType est abstrait et doit être concrétisé par les types correspondant aux NfpMeasures définies, la figure montre un exemple de deux types concrets : *Float* et *Integer*. À chaque définition de type, il est nécessaire d'associer les implémentations des opérations d'ordre et de composition spécifiées par sa NfpMeasure correspondante. L'instanciation d'un type en tant que valeur s'effectue grâce à la méthode *instanciate* que chaque nouveau type doit surcharger afin de construire une WeightValue appropriée en fonction d'une valeur donnée en chaîne de caractère (valeur de NfpMeasure issue du modèle RecoMARTE).

La figure 5.5 présente la partie outillage du métamodèle intermédiaire pour le calcul des poids de configurations. Elle contient les classes abstraites de base à étendre pour définir de nouvelles opérations sur ces types de poids. Les classes BinaryOperation et OrderOperation spécifient ainsi les prototypes d'opération à étendre, respectivement pour composer deux valeurs d'un même type (en prenant une valeur neutre pour remplacer une valeur non définie) et pour comparer deux valeurs d'un même type. Les classes BinaryOperationClient et OrderOperationClient permettent de définir génériquement des opérations, par exemple la notion d'addition en tant qu'opération de composition, puis de surcharger les méthodes *accept* prenant en paramètre un type de poids (une par type de poids défini), et retournant une opération concrète associée à ce poids, par exemple l'addition sur les entiers.

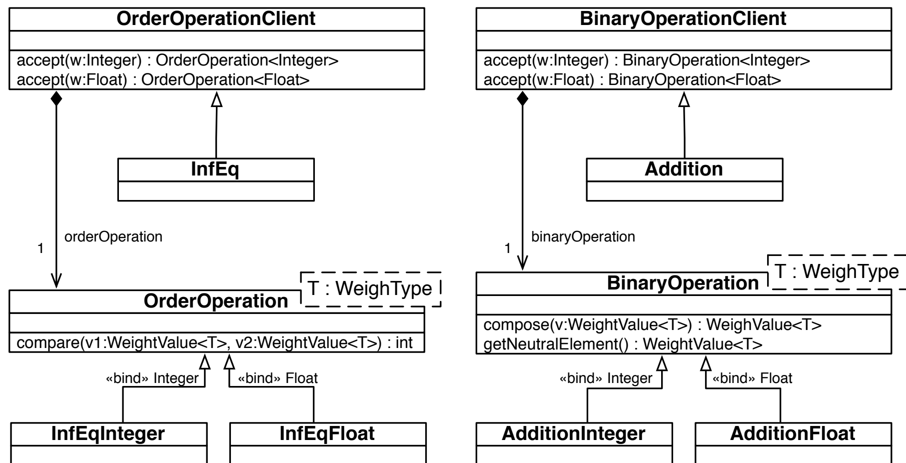


Figure 5.5 – Principe de définition des opérations sur poids dans le métamodèle intermédiaire

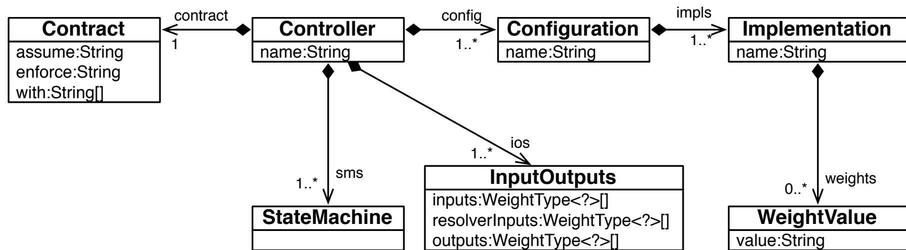


Figure 5.6 – Principe de définition des contrôleurs dans le métamodèle intermédiaire

Avec un tel métamodèle outillé, il devient possible d'automatiser le calcul des poids de configuration à partir de ceux issus de la combinaison de leurs implémentations. cette étape sera établie lors de la transformation vers BZR. À partir de l'ensemble des configurations, de leurs poids calculés, et de leurs opérations d'ordre ainsi définies, il sera ensuite trivial de classer les configurations dans un treillis d'après les valeurs de leurs poids.

5.2.3 Propriétés du contrôleur

Un modèle RecoMARTE peut être constitué de contrôleurs (RtUnits disposant du stéréotype «Controller»), lesquels sont associés avec un ensemble de ModeBehaviors, de NfpConstraints, de ports d'entrées/sorties, de types de NfpMeasures et de résolveurs de signaux internes. La représentation des contrôleurs, ainsi que leurs connexions avec les éléments précédents, est donnée par l'extrait du métamodèle intermédiaire en figure 5.6.

Ce qui est conservé pour chaque contrôleur RecoMARTE dans cette représentation intermédiaire est donc un composant (par contrôleur) muni d'un ensemble de StateMachines et Configurations qu'il pilote, d'entrées/sorties (InputOuputs) dont les types doivent éga-

lement être définis parmi les types de poids du métamodèle intermédiaire, et d'un élément nommé *Contrat* (Contract) regroupant les propriétés suivantes :

- *assume*, la composition par un "Et" booléen des éventuelles expressions booléennes définissant les règles à respecter pour l'utilisation du contrôleur (NfpConstraints de type "required") ;
- *enforce*, idem qu'*assume* mais pour les règles garanties par le contrôleur (NfpConstraints de types "contract") ;
- *with*, la liste des noms de variables (booléennes) internes au contrôleur.

Les ports d'entrées/sorties du contrôleur, encapsulés par l'élément *InputOutput*, sont distingués en trois catégories : 1) les entrées en direction des StateMachines, *inputs*, 2) les entrées réservées au resolver, *resolverInputs*, 3) et les sorties du contrôleurs, *outputs*. On notera que la notion de *resolver*, pourtant présente dans le modèle RecoMARTE, est ici abstraite. On ne conserve que son interface : ses ports de sorties sont représentés par la propriété *with*, et ses ports d'entrées sont agrégés dans la propriété *resolverInputs*.

Toutes ces informations ainsi extraites et structurées/calculées vont maintenant servir à construire une représentation synchrone en utilisant le langage BZR.

5.3 Implémentation exécutable du contrôle

5.3.1 Challenge de l'implémentation maximalement permissive

Pour rappel, le problème principal relevé dans le chapitre précédent concernant l'implémentation manuelle de l'algorithme de valuation des signaux internes est celui de l'implémentation sous-optimale. En effet, s'il reste possible pour le concepteur d'assurer – point important dans cette étude – la sécurité de son implémentation manuelle grâce à des outils de vérification, il ne peut assurer formellement que son implémentation est maximalement permissive sans un outil de synthèse approprié.

À ce titre, le modèle de Ramadge et Wonham [114] pose les fondamentaux de la synthèse du contrôle en montrant comment un contrôleur est capable d'inhiber des transitions d'un automate afin de garantir une spécification comportementale. Ce modèle sera par la suite amélioré, notamment dans [65], motivant l'emploi de méthodes symboliques (représentation d'espaces d'états par le biais de formules) pour l'optimisation de l'algorithme de synthèse.

Afin de comprendre l'apport de la synthèse de contrôleur pour traiter le problème de l'implémentation maximalement permissive, il convient d'introduire certaines notations sur lesquelles cette méthode s'appuie :

Soit S une vue logique du système reconfigurable considéré, modélisée à l'aide d'un

système de transition étiqueté, ou *Labelled Transition System* (LTS) (dans le cas présent : un ensemble d'automates). Cette modélisation est définie par le tuple suivant : $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ où $\mathcal{Q}$ est un ensemble fini d'états, q_0 est l'état initial de S, $\mathcal{I}$ est un ensemble fini d'événements d'entrée (fournis par l'environnement), $\mathcal{O}$ est un ensemble fini d'événements de sortie (émis vers l'environnement), et $\mathcal{T}$ est la relation de transition, c'est-à-dire un sous-ensemble de $\mathcal{Q} \times \text{Bool}(\mathcal{I}) \times \mathcal{O}^* \times \mathcal{Q}$, où $\text{Bool}(\mathcal{I})$ est l'ensemble des expressions booléennes sur $\mathcal{I}$. Soit $\mathcal{B}$ l'ensemble $\{\text{true}, \text{false}\}$, alors une garde $g \in \text{Bool}(\mathcal{I})$ peut être vue en tant que fonction de $2^{\mathcal{I}}$ vers $\mathcal{B}$.

Toute transition possède une étiquette de la forme g/a, où $g \in \text{Bool}(\mathcal{I})$ doit être vraie (true) pour que la transition associée puisse être prise (g est la *garde* de la transition), et où $a \in \mathcal{O}^*$ est une conjonction des signaux de sortie, émis lorsque la transition est prise (a est l'*action* de la transition). L'état q est la *source* de la transition (q, g, a, q'), et l'état q' est la *destination*. Une transition (q, g, a, q') se représente en notation graphique par $(q \xrightarrow{g,a} q')$.

L'ensemble des automates définis dans une spécification RecoMARTE peut se calquer sur un tel modèle. Le LTS désignant S est potentiellement composé de plusieurs automates (donc d'autres LTS) : leur éventuelle composition hiérarchique est considérée *à plat* (tout état d'un automate encapsulé dans un mode dispose d'une duplication des transitions sortantes de ce mode, le même principe s'applique récursivement si cet automate encapsule d'autres automates dans ses propres modes), et leur éventuelle composition parallèle est également *aplatie* par l'*opérateur de composition synchrone*.

L'opérateur de composition, noté $\|$, de deux LTS définis en parallèle désigne leur produit synchrone ; il s'agit d'une opération commutative et associative, commune à tout langage synchrone. Formellement :

$\langle \mathcal{Q}_1, q_{0,1}, \mathcal{I}_1, \mathcal{O}_1, \mathcal{T}_1 \rangle \| \langle \mathcal{Q}_2, q_{0,2}, \mathcal{I}_2, \mathcal{O}_2, \mathcal{T}_2 \rangle = \langle \mathcal{Q}_1 \times \mathcal{Q}_2, (q_{0,1}, q_{0,2}), \mathcal{I}_1 \cup \mathcal{I}_2, \mathcal{O}_1 \cup \mathcal{O}_2, \mathcal{T} \rangle$
avec $\mathcal{T} = \{((q_1, q_2) \xrightarrow{(\wedge_{k=1}^{n} g_k)/(\wedge_{k=1}^{n} a_k)} (q'_1, q'_2)) | (q_k \xrightarrow{g_k/a_k} q'_k) \in \mathcal{T}'_k, (q_k, q'_k) \in q \times q'\}$.

Ici (q_1, q_2) est appelé un *macro-état*, où q_1 and q_2 sont ses deux *sous-états*. Un macro-état contenant un sous-état par LTS composé de manière synchrone dans un système S est appelé une *configuration* de S. Cette définition coïncide avec la notion de configuration logique d'un système reconfigurable : la composition des états actifs de ce système dans un instant discret considéré.

Pour simplifier les explications, les éventuelles étapes de compression/optimisation (notamment celles employant une représentation par des BDD) ne sont pas abordées. Le principe de synthèse de contrôleur sera ici illustré directement sur une représentation à plat d'un LTS :

Soit S le LTS donné en figure 5.7. Donc $\mathcal{Q} = \{A, B, C, D, Err\}$, $q_0 = A$, $\mathcal{I} = \{u, c\}$, $\mathcal{O} = \emptyset$, et $\mathcal{T}$ regroupe l'ensemble des transitions, munies de leurs gardes. Maintenant

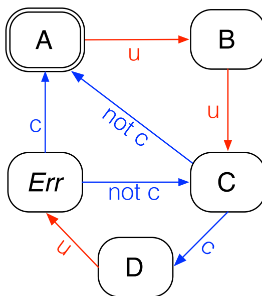


FIGURE 5.7 – Exemple de système à contrôler, représenté par un LTS

associons une propriété temporelle (expression booléenne) $\mathcal{F}$ telle que $\mathcal{F}$ doit être respectée pour toute exécution de S, avec $\mathcal{F} = not(Err)$, autrement dit : l'état Err ne doit jamais être actif. Définissons également une propriété temporelle $\mathcal{H}$, supposée être respectée par l'environnement, tel que $\mathcal{H} = true$, autrement dit ici : tout comportement de l'environnement est correct.

Partitionnons l'ensemble des entrées $\mathcal{I}$ en deux sous-ensembles : $\mathcal{I}_U$ l'ensemble des entrées en provenance de l'environnement, et $\mathcal{I}_C$ l'ensemble des entrées dont la valeur devra être fournie par un processus C dont le rôle est d'assurer $\mathcal{F}$ pour S sachant $\mathcal{H}$. Formellement : $\mathcal{I} = \mathcal{I}_C \cup \mathcal{I}_U$ et $\mathcal{I}_C \cap \mathcal{I}_U = \varnothing$, avec ici $\mathcal{I}_U = \{u\}$ et $\mathcal{I}_C = \{c\}$.

L'objectif de la synthèse de contrôleur est d'obtenir automatiquement le code de C, tel que C n'impose de valeur à une variable de $\mathcal{I}_C$ que lorsque cela est nécessaire pour garantir $\mathcal{F}$. Le fonctionnement de C consiste donc à restreindre les transitions de S, en désactivant – par valuation appropriée de $\mathcal{I}_C$ – celles qui compromettraient ou permettraient de compromettre $\mathcal{F}$. En conséquence, la garde $g \in \text{Bool}(\mathcal{I}_U \cup \mathcal{I}_C)$ d'une transition peut être vue en tant que fonction de $2^{\mathcal{I}}_U \times 2^{\mathcal{I}}_C$ vers $\mathcal{B}$.

Une transition est dite *contrôlable* si et seulement si il existe une valuation de $\mathcal{I}_C$ telle que sa garde g est fausse. Dans le cas contraire, elle est *incontrôlable*. Formellement, une transition $(q, g, a, q') \in \mathcal{T}$ est contrôlable si et seulement si $\exists X \in 2^{\mathcal{I}_C}$ tel que $\forall Y \in 2^{\mathcal{I}_U}$, alors $g(X \cup Y) = false$.

La synthèse de contrôleur, dans sa version simplifiée s'exerçant sur un LTS à plat muni de contraintes et de variables contrôlables, repose sur l'établissement de la région maximale d'états autorisés dans ce LTS. Ce principe est montré par le programme 5.

Exécutons ce programme sur le LTS donné en exemple. Au commencement, on établit la région d'états privée de ceux enfreignant la contrainte $\mathcal{F}$, en l'occurrence ici : *Err* cf. figure 5.8a). La corps de la boucle *while* du programme sert à rechercher parmi les états de cette région, ceux disposant d'une transition ciblant un état en dehors (donc

```
    input  : Q, q0, I_U, I_C, O, T, F, H
    output : La région maximale d'états autorisés
 1  begin
 2      R' ← Q;
 3      foreach q ∈ R' do
 4          if q ⇒ ¬F then
 5              R' ← R'\q ;
 6          end
 7      end
 8      R ← ∅;
 9      while R ≠ R' do
10          R ← R';
11          foreach q ∈ R' do
12              foreach q' ∉ R do
13                  if (q --g/a--> q') ∈ T then
14                      if ∃Y ∈ 2^{I_U}, Y ⇏ ¬H | ∀X ∈ 2^{I_C}, g(X ∪ Y) ≠ false then
15                          R' ← R'\q ;
16                      end
17                  end
18              end
19          end
20      end
21      if q0 ∈ R then
22          Print "Le système est contrôlable.";
23          return R ;
24      else
25          Print "Le système n'est pas contrôlable";
26          return null;
27      end
28  end
```

Algorithme 5: Extraction de la région d'états permis d'un système donné

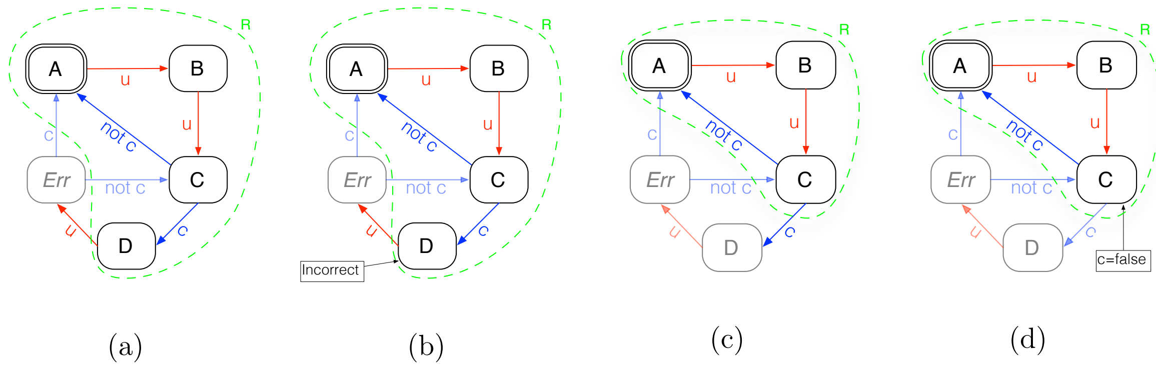


FIGURE 5.8 – Extraction de la région maximale d'état autorisés dans le LTS d'exemple

un état interdit). Si, pour un tel état, la transition concernée ne peut être contrôlée, alors l'état doit être interdit car une valuation légale d'événements en provenance de l'environnement peut déclencher cette transition sans qu'aucune valuation de variable interne ne puisse l'empêcher. À la première itération, c'est le cas de l'état D (cf. figure 5.8b), disposant d'une transition incontrôlable vers *Err*. D est donc retiré de R (cf. figure 5.8c). À l'itération suivante, l'état C dispose d'une transition vers D (donc en dehors de R), mais celle-ci est contrôlable. En effet, en valuant la variable contrôlable c à *false*, il est possible d'empêcher cette transition (cf. figure 5.8d). Enfin, à l'itération suivante, R converge, c'est-à-dire qu'aucun état n'est nouvellement interdit. La région R contient l'état initial A, cela signifie qu'à l'initialisation du LTS, celui-ci est bien dans un état autorisé. À partir de A, il est donc possible, sous contrôle, de rester dans la région R d'état autorisée, cette région étant maximale.

Il s'agissait ici de montrer le principe de base de la synthèse de contrôleur. Des méthodes de contrôle plus complexes existent, prenant par exemple en compte des poids associés aux états, ou la notion de *valeur à un instant précédent*. Il s'agit à chaque fois de pouvoir exprimer ce type de contrôle vers un LTS à plat (au moyen de duplication d'états pour représenter les divers poids possibles d'un états, ou des séquences de valeurs possibles), puis d'appliquer le même principe de recherche de région maximale (avec optimisation possible en passant par exemple par une représentation en BDD, comme c'est le cas dans SIGALI [71]).

Avec la connaissance de la région maximale R, le processus de contrôle C est capable de valuer correctement $\mathcal{I}_C$ pour tout instant donné d'exécution : si le système est dans un état en périphérie de R, C impose une valuation de $\mathcal{I}_C$ telle qu'aucune transition sortant de R ne peut être prise. Le détail de cette valuation sera vu plus loin.

5.3.2 Transformation vers BZR et exploitation de la synthèse de contrôleur

La modélisation à base de LTS précédemment définie pour la démonstration de synthèse de contrôleur (SDC) coïncide avec la représentation – extraite en modèle intermédiaire – des systèmes modélisables en RecoMARTE. Cette notation introduite sera ainsi réutilisée pour décrire des transformations. Il s'agit maintenant de donner la correspondance des modèles intermédiaires, par transformation, en langage BZR pour préparer l'étape de SDC. La syntaxe de BZR sera détaillée au fur et à mesure.

Nœud principal de la spécification du contrôle

Le premier élément à cibler en BZR est la notion de *nœud* [29]. Un nœud est un composant exécutable disposant d'un identifiant et définissant une interface composée d'événements d'entrée et de sortie. Il permet de déclarer un *contrat*, c'est-à-dire un ensemble d'événements internes (mot clé *with*), combiné à des contraintes temporelles d'invariance qui peuvent être soit garanties par ce nœud (mot clé *enforce*), soit supposées pour l'environnement (mot clé *assume*). Un exemple de déclaration de nœud avec contrat est donnée dans le programme n°6.

```
1  node (e1 : bool; e2 : int)
2  returns (o : int)
3  contract
4    assume (true)
5    enforce (not(StateA and StateB))
6    with (c1 : bool; c2 : bool)
7  var
8    (* Déclaration de variables internes *)
9  let
10   (* Déclaration (par équations) du corps du nœud *)
11 tel
```

Algorithme 6: Exemple de déclaration de nœud avec contrat

La correspondance avec un *Controller* depuis le modèle intermédiaire (cf. section 5.2) est immédiate. Soient *type* et *id* le type et l'identifiant d'un événement (entrée, sortie, contrôlable ou non), l'interface générique d'un nœud BZR correspondant à un contrôleur est définie dans l'extrait de programme n°7. À noter que la sortie $\mathcal{O}$ du contrôleur se décompose en deux parties : les éventuels événements de sorties arbitraires définis par le

concepteur, et la configuration active, donnée par un ensemble de booléens (un par état) dont ceux valués à *true* correspondent à la combinaison des états actifs dans l'instant courant.

```
1 node (id : type (* ∀id ∈ I_U *) )
2 returns (id : type (* ∀id ∈ O *) )
3 contract
4    assume (H)
5    enforce (F)
6    with (id : bool (* ∀id ∈ I_C *) )
```

Algorithme 7: Correspondance générique en BZR de l'interface d'un contrôleur

Automates

Le corps d'un nœud BZR peut s'exprimer par un ensemble d'équations synchrones, mais BZR dispose également d'une syntaxe d'abstraction permettant de représenter des automates de mode. C'est donc naturellement que les StateMachines du modèle intermédiaires vont se calquer sur cette représentation. Afin de connaître l'activation ou non d'un mode en BZR, il est possible de déclarer des équations appropriées dans chaque mode (celles-ci sont dont évaluées seulement lorsque leur mode est actif) afin de valuer des variables suivant la correspondance *une variable par mode*, telle que chaque variable associée à un mode est vraie seulement lorsqu'il est actif. Soit E l'ensemble des identifiants d'états déclarés, la partie *Déclaration de variables internes* de l'algorithme 8 montre la définition générique de ces variables. Soit α le symbole représentant une StateMachine, La structuration en automates BZR des StateMachines du modèle intermédiaire suit le modèle parallèle et hiérarchique spécifié dans la partie *Implémentation du corps du nœud* de l'algorithme 8.

Configurations

Pour finir, les configurations du système ainsi que leurs poids à calculer doivent être représentés dans la génération du programme correspondant en BZR. Il s'agit de définir une variable par type de poids déclaré par le contrôleur, et de calculer leur valeur à partir de l'information des états actifs : si un état est actif, sa valeur associée pour ce type de poids doit être composée avec celle des autres états actifs.

Formellement, soit W l'ensemble des types de poids déclarés. Tout poids w de type f, noté w_f tel que $w_f \in W$ est associé à un ensemble symbolisé par V_f, ainsi qu'à une

```
1  ... (* Interface du nœud *)
2  var
3    (* Déclaration de variables internes *)
4    (b_s : bool) (* ∀b_s ∈ E *)
5  let
6  |  (* Implémentation du corps du nœud *)
7  |  automaton (* ∀α ∈ S *)
8  |  |  state s (* ∀s ∈ α *)
9  |  |  |  do
10 |  |  |  b_s' ← (s' = s) ; (* ∀s' ∈ α *)
11 |  |  |  ... (* Autres équations arbitraires *)
12 |  |  |  automaton (* ∀α ∈ s *)
13 |  |  |  |  ... (* Définition récursive de l'éventuelle composition hiérarchique
   |  |  |  |  d'automates *)
14 |  |  |  end ;
15 |  |  |  unless g_α then s' (* g_α ∈ g, s ∈ q, s' ∈ q', (q --g/a--> q') ∈ T *)
16 |  |
17 |  end ;
18 tel
```

Algorithme 8: Déclaration générique d'automates et de variables internes liées à l'activation d'états

opération de composition sur cet ensemble notée $\star_f$ (disposant d'un élément neutre : $\varnothing_f$), et à une relation d'ordre sur cet ensemble désignée par $\leq_f$. Soit n_k le nombre d'états d'une configuration donnée, $0 \leq k < n_k$, q_k un état d'une configuration, et b_{q_k} la valeur booléenne symbolisant l'activation de l'état q_k. (Les b_{q_k} correspondent aux identifiants de E, cf. algorithme 8). Si q est la configuration courante, alors la valeur de tout poids v de type f – noté v_f de type w_f tel que $v_f \in V_f$ – se calcule par la combinaison des valeurs associées v_{f_k} de chaque état q_k de q :

$$\forall w_f \in W, v_f = \{\bigstar_{f\,k=1}^{m} v_{f_k} \mid b_{q_k}\}, \tag{5.1}$$

$$b_{q_k} = \begin{cases} true & \text{si } q_k \in q \\ false & \text{sinon} \end{cases}$$

Les poids étant définis statiquement, les opérations de compositions peuvent être calculées hors ligne. L'association des valeurs calculées en tant que poids de la configuration active relève par contre de la responsabilité du contrôleur. En BZR, le programme cible doit donc disposer des valeurs calculées pour toute configuration, ainsi que d'équations pour les associer aux variables symbolisant les poids de la configuration active. Ceci est montré génériquement par l'algorithme 9, où n représente le nombre de configurations du système S, $0 \leq i < n$, q_i est la configuration d'identifiant i, et v_{f_i} est la valeur du poids de type w_f pour q_i. Cela autorise ainsi toute équation, en particulier celles définies en tant que contraintes ($\mathcal{F}$ et $\mathcal{H}$), à référencer les valeurs v_f des poids de la prochaine configuration active dans le système.

5.4 Problème du non déterminisme de la représentation en automates

La compilation d'un nœud BZR correspondant à la description générique qui vient d'être présentée produit, si la synthèse de contrôleur termine correctement, un programme dont l'exécution repose sur le principe des systèmes réactifs contrôlés présentés dans l'algorithme 4 du chapitre 3. Ce programme reçoit, à chaque itération de sa boucle d'exécution, l'ensemble des événements définis en entrée du nœud et produit l'ensemble des événements de sortie également définis. L'exécution assure, pour toute itération, que les contraintes spécifiées dans $\mathcal{F}$ seront assurées tant que les contraintes données par $\mathcal{H}$ sont respectées par l'environnement. Afin de garantir $\mathcal{F}$, le programme donne une valeur appropriée aux variables spécifiées dans $\mathcal{I}_C$: les variables contrôlables.

```
1  ... (* Interface du nœud *)
2  var
3     (* Déclaration de variables internes *)
4     (* Dont les variables symbolisant l'activation des états : *)
5     (b_{q_k} : bool) (* ∀b_{q_k} ∈ E *)
6     (* Ainsi que les variables de poids de la configuration active...*)
7     (v_f : w_f) (* ∀f|w_f ∈ W *)
8     (*... et celles des poids des configurations *)
9     (v_{f_i} : w_f) (* ∀v_{f_i} ∈ V_f, 0 ≤ i < n *)
10 let
11    (* Déclaration (par équations) du corps du nœud *)
12    (* Automates, équations arbitraires, etc. *)
13    v_f = Σ_{i=0}^{n} v_{f_i} (* ∀f|w_f ∈ W *)
14    v_{f_i} = { ★_{f, k=1}^{m} v_{f_k}   si {∧ b_{q_k} | q_k ∈ q_i}
              {                0   sinon
15 tel
```

$$v_f = \sum_{i=0}^{n} v_{f_i}$$

$$v_{f_i} = \begin{cases} \bigstar_{f\,k=1}^{m} v_{f_k} & \text{si } \{\bigwedge b_{q_k} \mid q_k \in q_i\} \\ 0 & \text{sinon} \end{cases}$$

Algorithme 9: Valuation des poids v_f de la configuration active en fonction des valeurs v_{f_i} des poids des configurations.

La méthode de valuation des variables contrôlables est dite *maximalement permissive*. Cela signifie que la fonction de valuation, construite par le processus de synthèse de contrôleur, ne donne pas de valeur à une variable contrôlable si sa valuation par *true* ou par *false* ne compromet (et ne compromettra) pas $\mathcal{F}$. Une telle variable est dite *libre*. Autrement dit, le *contrôleur* produit par la synthèse de contrôleur n'est pas déterministe dans le cas général, et n'est donc pas exécutable. Le compilateur BZR vient outiller ce contrôleur par un mécanisme de valuation supplémentaire, reposant sur les *oracles* présentés dans [4].

L'objectif de cette section est de montrer en détails le mécanisme complet de valuation des variables contrôlables dans BZR, d'en relever les problèmes, et de spécifier les besoins pour pouvoir s'adapter.

5.4.1 Exemple

La valuation des variables contrôlables libres via la méthode d'oracles repose sur le principe donné dans l'algorithme 10. Les oracles représentent un vecteur $\mathcal{L}$ de valeur booléennes, défini statiquement, tel que $\text{card}(\mathcal{L}) = \text{card}(\mathcal{I}_C)$. À chaque itération principale de cet algorithme (à chaque tick), les variables contrôlables sont traitées une par une, le contrôleur (appelé par la fonction *ComputeControllable*) calcule la valeur d'une de ces variables ou la laisse libre si elle peut être vraie ou fausse ; si elle est libre, elle prend alors la valeur correspondante donnée dans le vecteur d'oracles ; et ainsi de suite jusqu'à ce que toutes les variables contrôlables soient traitées dans une itération, auquel cas le programme peut mettre à jour son état et produire des événements de sortie. Un vecteur d'oracles tous valués à *true* est généré par défaut dans la compilation d'un programme BZR, donc par défaut toute variable contrôlable libre est valuée à *true*.

La valuation par oracles, qu'elles soit définie statiquement comme c'est le cas par défaut, ou bien dynamiquement au moyen d'un algorithme de valuation en ligne pose exactement le même domaine de problèmes évoqué lors de la réalisation manuelle du contrôleur :

- une valuation statique risque d'être sous-optimale ;
- une valuation dynamique demande de calculer en ligne les combinaisons légales de valuations de variables contrôlables afin d'en choisir une, ce qui représente un calcul complexe et peu approprié dans un cadre temps-réel.

Pour illustrer ces deux types de problèmes, repartons sur les deux machine à état présentées en figure 4.16 du chapitre précédent, et admettons que le programme BZR équivalent soit synthétisé, en prenant en compte la contrainte temporelle de non-activation simultanée de B et D.

Supposons que le système soit en configuration {A ;D}, et que les événements *goB*, *goC*

```
1  begin
2    state ← initial ;
3    foreach tick do
4      Read(I_u);
5      Valued ← ∅ ;
6      foreach c_i ∈ I_C (* 0 ≤ i < card(I_C) *) do
7        c_i ← ComputeControllable(I_u, Valued, state);
8        if isFree(c_i) then
9          c_i ← l_i ; (* l_i ∈ L *)
10       end
11       Valued ← Valued + c_i;
12     end
13     (output,state) ← f(I_u, I_c, state);
14     Write(output);
15   end
16 end
```

Algorithme 10: Exécution d'un système réactif sous contrôle maximalement permissif avec méthode de valuation complémentaire à bas d'*oracles*

et *goD* prennent respectivement les valeurs *true*, *false* et *false*. Maintenant admettons que nous ayons décidé statiquement de poser les valeurs d'oracles *true* et *false* respectivement pour les deux variables contrôlables $c1$ et $c2$. Pour l'instant, tant que $c1$ et $c2$ ne sont pas valuées, ces deux variables sont libres.

Cas 1 : Si $c1$ est valuée en premier, elle prend donc la valeur de son oracle : *true*, mais dans ce cas, $c2$ n'est plus libre et doit prendre la valeur *true* imposée par le contrôleur afin de forcer la transition de D vers C (et ainsi éviter d'activer en même temps B et D).

Cas 2 : Si $c2$ est valuée en premier, elle prend donc la valeur de son oracle : *false*, mais dans ce cas, $c1$ n'est plus libre et doit prendre la valeur *false* imposée par le contrôleur afin d'interdire la transition de A vers B (et ainsi éviter d'activer en même temps B et D).

Nous avons ici un cas de figure pour lequel la priorité arbitraire de valuation des variables contrôlables influence la propriété *libre* ou non des autres variables contrôlables. Selon le contexte d'exécution, il se peut que la valuation $c1 = true, c2 = true$ soit plus *optimale* dans certaines itérations de l'exécution du système que $c1 = false, c2 = false$. Or la définition arbitraire des oracles (et de la priorité de valuation) impose d'effectuer un choix statique, donc potentiellement sous optimal dans certains contextes d'exécution.

Si la valuation est dynamique, il est alors nécessaire de tester les combinaisons de valuation avant de les imposer. Ici, il faudrait tester (en appelant le contrôleur) les combinaisons 1) $(c1 = true, c2 = true)$, 2) $(c1 = true, c2 = false)$, 3) $(c1 = false, c2 = true)$ et 4) $(c1 = false, c2 = false)$. Le contrôleur rejèterait alors les combinaisons 2 et 3 ; il resterait alors au mécanisme de valuation dynamique d'effectuer un choix entre 1 et 4, choix qui pourrait s'effectuer de manière optimale selon le contexte d'exécution. On comprend que cette méthode de valuation est complexe, puisque qu'elle demande de réaliser des calculs exponentiels (appel de la fonction de contrôle à chaque test de combinaison de valuation).

Proposer une méthode (heuristique ou exacte) afin d'éviter un calcul exhaustif en ligne des combinaisons n'est pas non plus une solution pour l'approche de valuation dynamique : en effet toute valuation n'étant pas nécessairement une solution légale, la mise en œuvre d'une méthode admissible demanderait d'établir une information comportementale du contrôleur ; or l'espace des solutions de valuations ne dispose pas dans le cas général d'une structure régulière, et dépend en plus de l'état courant du système et des événements en entrée. Une approche complètement dynamique demanderait donc, paradoxalement, de tester à chaque itération toutes les combinaisons de valuations en fonction de l'état courant et des entrées afin d'en rechercher la structure, ce qui n'est bien évidemment pas une solution comparée à l'approche exhaustive. Si l'établissement des valuations possibles est effectué statiquement afin d'optimiser les reconfigurations, c'est-à-dire que pour tout état du contrôleur, on recherche, pour toutes entrées possibles, les meilleures solutions

de valuations, on rentre alors dans le registre de la *synthèse de contrôleur optimale* qui consiste à garantir un chemin optimal de reconfiguration sur N itérations à l'avance [34]. Un tel contrôleur *optimal* s'affranchit des oracles en imposant une valuation *optimale sur N itérations* des variables libres, il n'est donc plus maximalement permissif, dans le sens où il ne permet pas de tendre vers une solution de valuation sous-optimale sur N itérations.

Ce cadre d'utilisation de la synthèse de contrôleur est en cours d'étude par un autre acteur du projet FAMOUS [6], notamment pour optimiser les choix d'ordonnancements d'une séquence tâches.

La présente étude propose d'aborder une autre approche du problème, qui consisterait non pas à explorer un espace de valuation, mais à disposer d'un espace de configurations possibles, directement fourni par le contrôleur. S'il ne s'agit pas ici d'aller à l'encontre du mécanisme de valuation statique imposé dans BZR, il s'agit cependant de tenter d'optimiser les choix de reconfiguration lorsque, pour une valuation de variables données – contrôlables et non contrôlables –, plusieurs reconfigurations sont possibles dans un instant considéré.

5.4.2 Différence sémantique entre les ModeBehavior et les automates en BZR

Une propriété intéressante des ModeBehaviors, ainsi que des UML State Machines, est leur absence de déterminisme. Lorsque, depuis un mode actif, deux transitions sortantes (vers deux modes différents) ont leur garde évaluée à *true*, la sémantique suppose que le concepteur dispose d'un mécanisme de décision pour gérer ce cas de figure et opérer un choix [124].

En BZR, les automates spécifiés sont déterministes, un tel choix existe donc pour traiter cette ambiguïté et est défini statiquement : l'ordre de spécification des transitions dans le programme BZR définit leur priorité d'évaluation. Donc si deux transitions sortantes d'un même état sont vraies, la première a avoir été spécifiée est prise.

Encore une fois, ce cadre de décision défini statiquement est potentiellement sous-optimal. Selon le cas d'exécution, il peut tout à fait être plus intéressant de prendre la seconde transition, ce qui serait tout à fait acceptable dans la sémantique du ModeBehavior correspondant.

Les automates dans BZR ne sont cependant qu'une syntaxe particulière, définie à juste titre pour son côté *user friendly*. Ils pourraient tout à fait être spécifiés sous forme équationnelle : dans un système d'équations synchrone, les équations sont toutes évaluées en concurrence, cela signifie que si un automate est représenté sous forme équationnelle,

deux gardes ayant les propriétés que nous avons mentionnées seront évaluées quelle que soit leur valeur et sans question de priorité empêchant l'évaluation de l'autre.

5.4.3 Exploitation du non-déterminisme pour la décision en ligne de reconfiguration

Il semble manifestement plus intéressant de transformer les ModeBehaviors vers une représentation équationnelle plutôt que vers des automates dans la sémantique imposée par BZR, si l'on souhaite optimiser en ligne les choix de reconfiguration en se basant sur l'accessibilité d'états.

L'idée serait en effet de pouvoir profiter de la sémantique indéterministe dans MARTE, afin de cibler une représentation équivalente en BZR pour laquelle chaque nœud (RtUnit de contrôle) permettrait, sur occurrence des événements spécifiés dans le modèle MARTE, de produire non pas un choix de configuration mais un ensemble de configurations calculées comme légales à la prochaine itération. Un mécanisme externe, que le concepteur pourrait implémenter arbitrairement, viendrait alors sélectionner l'une d'entre elles, et cette sélection ferait ainsi office d'ordre de reconfiguration pour le système.

L'avantage d'une telle approche serait de pouvoir disposer du mécanisme simple de valuation statique (déterministe) des variables contrôlables, en employant les oracles, tout en disposant d'une exécution du contrôle non-déterministe permettant d'implémenter une fonction de détermination arbitraire à des fins d'optimisation : la *décision*.

Concevoir une spécification de contrôle afin de tirer parti au mieux de cette approche, puis la transformer vers une représentation BZR équivalente autorisant l'implémentation d'un mécanisme de décision fait l'objet du prochain chapitre.

5.5 Conclusion

Ce chapitre a présenté la méthodologie de transformation d'une spécification RecoMARTE vers une spécification BZR apte à être synthétisée en vue d'obtenir, pour chaque RtUnit de contrôle défini, un contrôleur de reconfiguration. Cette transformation a nécessité l'établissement d'un modèle intermédiaire qui deux objectifs :

- donner une représentation objet la plus proche possible de la cible (code BZR), afin de simplifier la transformation finale de type *modèle vers texte*.
- implémenter concrètement les opérations sur les poids de configurations (valeurs de NfpMeasures), afin d'automatiser leur calcul.

L'implémentation exécutable du contrôle, donnée en BZR, permet de répondre au besoin de génération automatique et correcte du contrôleur – grâce au mécanisme formel

de synthèse de contrôleur incorporé dans le compilateur BZR –, l'implémentation manuelle ayant en effet été identifiée comme potentiellement complexe (et non sûre si elle n'est pas couplée à une vérification formelle).

Enfin, l'implémentation proposée est potentiellement optimisable, mais demande pour cela d'adapter les transformations afin que depuis une même spécification RecoMARTE il puisse être généré une représentation de contrôle équivalente, dans laquelle chaque contrôleur raisonne non plus sur la prochaine configuration à imposer au système, mais sur un ensemble de configurations autorisées. Cet ensemble pourra ensuite être fourni à un mécanisme de décision, dont le rôle est d'en choisir une, et de l'imposer au système. C'est ce que nous proposons maintenant d'aborder.

6

Combinaison du contrôle discret et de l'optimisation sous contrôle

Sommaire

6.1 Introduction **125**
6.2 Notion de décision après contrôle **126**
6.2.1 Nécessité de la décision en ligne 126
6.2.2 Représentation équationnelle non déterministe des automates 127
6.2.3 Problématique de la prise en compte du mécanisme de décision dans la spécification de contrôle 128
6.3 Formalisation du problème et adaptation des transformations **128**
6.3.1 Définitions 128
6.3.2 Adaptation des transformations 129
6.4 Exemple de décision générique **132**
6.4.1 Frontière de Pareto dynamique 134
6.4.2 Régulation par contrôle PI 135
6.4.3 Élection d'une configuration 139
6.5 Conclusion **140**

6.1 Introduction

Dans le chapitre précédent, nous avons émis l'idée que l'implémentation d'un mécanisme de décision après contrôle, à la charge du concepteur, pourrait être utile afin de mieux maîtriser les possibilités de reconfigurations, au lieu de subir la rigueur d'une spécification statique. Pour cela, nous disposons d'un créneau à exploiter : le non-déterminisme

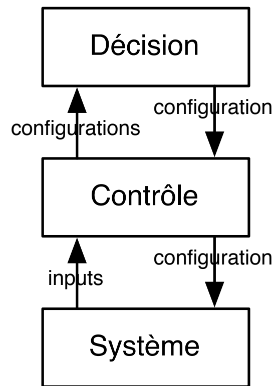


FIGURE 6.1 – Principe d'intégration contrôle/décision

de la spécification MARTE. La détermination arbitraire introduite dans les représentations intermédiaires, notamment la représentation synchrone en BZR, n'est en effet pas du ressort du concepteur. Autrement dit, pour une spécification comportementale donnée en MARTE, il existe plusieurs exécutions de contrôle correspondantes.

Il est proposé ici d'exploiter dynamiquement cette multiplicité à des fins d'optimisation. La solution avancée, cf. figure 6.1 repose sur la séparation des notions de *contrôle* et de *décision*. Le *contrôle* désigne ici l'établissement, à chaque itération de la boucle principale d'exécution d'un contrôleur, d'un ensemble de configurations vers lesquelles il est légal de transiter d'après l'état courant, les valeurs d'entrées et les valeurs de variables contrôlables, en respect des objectifs de contrôle et hypothèses comportementales sur l'environnement. La *décision* désigne le processus qui, prenant cet ensemble en entrée à chaque itération, désigne une et une seule configuration issue de l'ensemble. Le contrôleur de reconfiguration doit ensuite émettre ce choix vers le système afin qu'il se reconfigure.

L'objectif de ce chapitre est d'abord de détailler cette notion de décision après contrôle, et d'en mesurer les impacts sur le mécanisme original de contrôle présenté au chapitre précédent. Il s'agira ensuite de formaliser le problème et d'adapter en conséquence les transformations proposées jusqu'ici. Enfin, à titre d'exemple, il sera proposé de reprendre un mécanisme de décision déjà éprouvé dans le domaine des architectures reconfigurables afin de montrer comment l'adapter à la notion de contrôle présentée dans cette étude.

6.2 Notion de décision après contrôle

6.2.1 Nécessité de la décision en ligne

L'avantage des mécanismes d'optimisation statiques – ce qui concerne notamment la synthèse de contrôleur optimale – est d'employer un précalcul des possibilités de recon-

figurations afin de garantir une séquence de reconfiguration optimale au pire cas sur un nombre déterminé N d'itérations de contrôle (ou *pas de contrôle*).

Si la taille du système est *raisonnable* et si N est *petit*[31], alors la synthèse optimale est un choix intéressant. Par exemple dans le cas de FAMOUS, le choix d'ordonnancement d'une séquence de tâches est un cas soutenable pour la synthèse optimale : N peut être borné facilement puisqu'il s'agit du nombre de tâches dans la séquence, le cycle des reconfigurations reste alors petit si le nombre de tâches est peu élevé.

Pour les autres types de problèmes où aucune borne cyclique raisonnable n'existe, l'emploi de la synthèse optimale nécessite de donner une taille N arbitraire, ce qui est potentiellement sous-optimal. Il est dans ce cas préférable de se tourner vers des approches plus dynamiques qui, si elles ne peuvent offrir de garanties d'optimisation (exécution au pire cas, etc.), permettent (si le concepteur l'implémente ainsi) de mettre en œuvre des algorithmes de type *apprentissage*, *recherche opérationnelle*, etc. Autrement dit, ce sont des techniques qui peuvent apporter un plus en terme d'optimisation, dans les cas où la complexité combinatoire d'une approche statique deviendrait trop importante au point de rendre celle-ci inapplicable.

6.2.2 Représentation équationnelle non déterministe des automates

Nous avons vu au chapitre précédent que la sémantique d'automates dans BZR est déterministe, ce qui correspond donc à une instance particulière et arbitraire de la sémantique non-déterministe donnée en MARTE par le concepteur. Or cette syntaxe peut être exprimée sous forme équationnelle, ce qui, d'après ce qui a été dit, permettrait d'évaluer toutes les gardes des transitions sortantes d'un état courant, sans pour autant impliquer de choisir arbitrairement la première évaluée à *true* dans leur l'ordre de spécification.

Pour autant, une sémantique d'exécution comportementale déterministe doit être représentée dans la spécification synchrone, afin que la synthèse de contrôleur puisse disposer de ce modèle sur lequel elle effectue une exploration. C'est pourquoi la spécification de contrôle doit disposer d'une double représentation comportementale : l'une, déterministe et à base d'automates BZR, afin de représenter la configuration active du système, et l'autre, non-déterministe et sous forme équationnelle, servant à valuer un ensemble de prochaines configurations accessibles.

31. La complexité combinatoire de la synthèse de la synthèse optimale réserve l'approche à des problèmes de taille peu élevée (de l'ordre d'une centaine de milliers états si la synthèse est effectuée par une machine puissante).

6.2.3 Problématique de la prise en compte du mécanisme de décision dans la spécification de contrôle

Ces deux représentations doivent être synchronisées : la configuration active du système dans un pas de contrôle donné doit faire partie des propositions de configurations issues du pas précédent. Or cette synchronisation est effectuée en dehors de la spécification de contrôle, par le module de décision à implémenter. Cela pose naturellement un problème pour la synthèse de contrôleur qui doit alors disposer d'une visibilité sur les traitements de l'environnement, car sans cela, il ne lui est pas possible d'établir le lien logique s'effectuant entre le choix effectif de configuration et l'ensemble des choix possibles prévus à l'instant précédent.

Puisque la détermination de la reconfiguration pour un nœud donné est externalisée, il est alors nécessaire d'employer le mécanisme d'hypothèse, prévu dans le contrat BZR de ce nœud. L'idée est de parvenir à spécifier l'hypothèse qu'à chaque pas, *une et une seule* configuration est donnée en entrée du nœud, et cette configuration doit avoir été autorisée au pas précédent. Le cas particulier étant celui du premier pas d'exécution, pour lequel il n'y a pas encore eu de proposition de configurations : la toute première *décision* de reconfiguration doit alors coïncider avec la configuration initiale spécifiée dans MARTE (c'est-à-dire la combinaison des états initiaux dans les ModeBehaviors).

6.3 Formalisation du problème et adaptation des transformations

6.3.1 Définitions

Nous reprenons ici de la notation formelle proposée au chapitre précédent. Nous avons dit que la spécification BZR obtenue jusqu'ici par transformation est une instance particulière de la spécification MARTE donnée par le concepteur. Autrement dit, il existe un ensemble d'instances correctes pour une même spécification MARTE, c'est-à-dire un ensemble de systèmes *équivalents* dont la sémantique d'exécution reste – pour toute exécution – dans les bornes imposées par la spécification non-déterministe MARTE, en ne choisissant à chaque pas qu'une transition dite *potentielle*, caractérisant l'état ciblé comme *accessible*. Définissons formellement ces notions de *transition potentielle*, *équivalence entre systèmes* et *état (ou configuration) accessible*.

Une transition est dite *potentielle* si l'état courant du système est sa source, et si sa garde est évaluée à *true* dans le pas courant.

Deux systèmes S et S' partageant les même états et le même état initial sont dits

équivalents par rapport à un objectif $\mathcal{F}$ et une hypothèse $\mathcal{H}$ qu'ils ont en commun si, et seulement si, pour toute exécution légale d'après $\mathcal{H}$:

- l'objectif $\mathcal{F}$ est garanti dans S et dans S' ;
- toute transition potentielle à chaque pas d'exécution pour un état et des entrées données dans un système est également potentielle pour l'état réciproque et les mêmes valeurs pour leurs entrées communes dans l'autre système.

Deux systèmes équivalents sont donc des instances d'une spécification de système reposant sur les mêmes états, objectifs, et hypothèses, ayant des entrées communes et définissant l'ensemble des transitions potentielles à chaque pas d'exécution. Ils diffèrent seulement sur le choix final de transition, effectué à chaque pas parmi les transitions potentielles.

Enfin, un état dans un LTS est dit *accessible* si au moins une des transitions sortant de l'état courant vers cet état est potentielle (ce qui inclut par défaut le cas où cet état est l'état courant et aucune transition sortante n'est potentielle). Une configuration, ou *macro-état* composé sur l'ensemble des automates d'un LTS, est accessible si chaque état de sa composition est lui-même accessible.

6.3.2 Adaptation des transformations

Soit $S = \langle \mathcal{Q}, q_0, \mathcal{I}, \mathcal{O}, \mathcal{T} \rangle$ un système défini en tant que LTS ou en tant que composition de plusieurs LTS, et $\mathcal{F}$ son objectif de contrôle à garantir étant donné $\mathcal{H}$ l'hypothèse comportementale de l'environnement du système. L'objectif est de construire un système $S' = \langle \mathcal{Q}, q_0, \mathcal{I}', \mathcal{O}', \mathcal{T}' \rangle$, intégrant un module de décision effectuant à chaque pas un choix de configuration parmi un ensemble de configurations accessibles. S étant la représentation formelle d'une spécification de contrôle en MARTE, S' doit être équivalent à S vis-à-vis de $\mathcal{F}$ et $\mathcal{H}$, et sur les entrées/sorties communes $\mathcal{I}$ et $\mathcal{O}$, afin d'être une instance acceptable de cette spécification de contrôle.

Modification de l'interface

S' encapsule donc les entrées et sorties de S ($\mathcal{I}$ et $\mathcal{O}$), et les étend afin de prendre en compte un choix de configuration en entrée, et une proposition de configurations accessibles en sortie. Pour rappel, $\mathcal{I} = \mathcal{I}_U \cup \mathcal{I}_C$ donc $\mathcal{I}'$ dispose également des mêmes variables contrôlables. Soit n le nombre de configurations donné par l'ensemble des combinaisons d'états de S (ou de S', puisqu'il s'agit de conserver le même espace d'états), et soit m le nombre de LTS de S. On pose $0 \leq i < n$ et $0 \leq j < m$. Soient $\mathcal{D}$ et $\mathcal{P}$ deux ensembles de booléens, de cardinalité $\text{card}(\mathcal{D}) = \text{card}(\mathcal{P}) = n$, où $d_i \in \mathcal{D}$ correspond au choix retourné par le module de décision dans lequel une seule configuration est donnée à chaque pas. Formellement : $\forall d_i \in \mathcal{D}, \{d_i \leftarrow (i = x)\}$, avec $0 \leq x < n$, où x est l'identifiant de configuration

choisie, et $p_i \in \mathcal{P}$ reflète l'accessibilité de la configuration dont l'identifiant est i. Au moins une configuration doit être accessible à chaque pas, formellement : $\exists p_i \in \mathcal{P}, p_i = true$. Les entrées et sorties $\mathcal{I}'$ et $\mathcal{O}'$ de $\mathcal{S}'$ sont définies par :

$$\begin{aligned} \mathcal{I}' &= \mathcal{I} \cup \mathcal{D}, \text{avec } \mathcal{I} \cap \mathcal{D} = \varnothing \\ \mathcal{O}' &= \mathcal{O} \cup \mathcal{P}, \text{avec } \mathcal{O} \cap \mathcal{P} = \varnothing \end{aligned} \tag{6.1}$$

Chaque booléen p_i est défini par une équation évaluant la potentialité de toute transition ciblant un état de la configuration i. Cette équation est vraie si tous les états concernés sont accessibles. Soit s_{i_j} un booléen reflétant l'accessibilité de l'état j d'une configuration i, p_i est alors défini par l'équation suivante :

$$p_i = \bigwedge_{j=1}^{m} s_{i_j} \tag{6.2}$$

avec chaque s_{i_j} défini par :

$$\begin{aligned} s_{i_j} &= \bigvee\{(g \wedge b_q) \mid (q \xrightarrow{g/a} q') \in \mathcal{T}\} \\ b_q &= \begin{cases} true & \text{si } q \text{ est l'état courant} \\ false & \text{sinon} \end{cases} \end{aligned} \tag{6.3}$$

Afin d'assurer à chaque pas qu'au moins une configuration accessible est autorisée d'après $\mathcal{F}$, cette propriété doit être ajoutée en tant qu'objectif de contrôle afin que la synthèse de contrôleur puisse la garantir. Ainsi, l'objectif de contrôle $\mathcal{F}'$ de $\mathcal{S}'$ est donné par :

$$\mathcal{F}' = \bigvee_{i=1}^{n} p_i \wedge \mathcal{F}, p_i \in \mathcal{P} \tag{6.4}$$

Il est à noter que si $\mathcal{F}'$ est garanti, alors $\mathcal{F}$ est garanti implicitement pour $\mathcal{S}'$, ce qui est nécessaire afin que $\mathcal{S}'$ soit équivalent à $\mathcal{S}$ d'après $\mathcal{F}$.

Hypothèse d'une décision *correcte* en entrée

Le moyen d'émettre les prochaines configurations accessibles venant d'être spécifié, le comportement générique du choix final de configuration doit être défini. Ce choix, provenant d'un module externe à la spécification de contrôle, est assimilé à une *boîte noire* du point de vue de la synthèse de contrôleur : seules les entrées $\mathcal{P}$ et les sorties $\mathcal{D}$ de ce module sont connues pour l'instant car elles sont respectivement encapsulées dans les sorties et entrées de S'. Cependant, il est nécessaire de spécifier une hypothèse comportementale sur ce module, afin que la synthèse de contrôleur ait connaissance du lien logique unissant $\mathcal{P}$ et $\mathcal{D}$:

– au moins une configuration accessible, d'après la valeur de $\mathcal{P}$ au pas précédent, doit être fournie en entrée (au moins un booléen de $\mathcal{D}$ doit être valué à *true*) ;
– une seule configuration au maximum doit être choisie (un seul booléen de $\mathcal{D}$ doit être valué à *true*).

L'hypothèse $\mathcal{H}'$ de $\mathcal{S}'$ étend l'hypothèse $\mathcal{H}$ de $\mathcal{S}$ afin d'encapsuler les propriétés comportementales qui viennent d'être mentionnées. L'opérateur *pre*, tel qu'il est défini dans les langages synchrones, va ici permettre de raisonner sur les valeurs de $\mathcal{P}$ – les configurations accessibles – dans un pas précédent. Formellement, soient *atLeastOne* et *atMostOne* deux équations définissant respectivement 1) le fait qu'au moins une configuration accessible soit choisie et 2) qu'une seule configuration soit choisie :

$$\mathcal{H}' = \mathcal{H} \wedge atLeastOne \wedge atMostOne \tag{6.5}$$

Ainsi, $\mathcal{H}$, *atLeastOne* et *atMostOne* sont supposées vraies pour toute exécution. Par extension, si $\mathcal{H}'$ est vraie, alors $\mathcal{H}$ est vraie, ce qui est nécessaire afin que $\mathcal{S}'$ soit équivalent à $\mathcal{S}$ d'après $\mathcal{H}$. Soit z l'identifiant de la configuration initiale ($0 \leq z < n$), $d_i \in \mathcal{D}$ et $p_i \in \mathcal{P}$, *atLeastOne* est défini par :

$$\begin{aligned} atLeastOne \quad = \quad & ((true \rightarrow pre(p_z)) \wedge d_z) \vee \\ & (\bigvee((false \rightarrow pre(p_i)) \wedge d_i)), i \neq z \end{aligned} \tag{6.6}$$

La notation $z = x \rightarrow pre(y)$ signifie que z prend la valeur de y calculée au pas précédent, sauf pour le premier pas qui est un cas particulier (y n'a pas de valeur précédente) où z prend la valeur de x. Dans le cas de l'équation *atleastOne*, au premier pas la valeur de d_z doit nécessairement être *true* en raison de la valuation (par initialisation) à *true* de l'accessibilité de la configuration z, et de la valuation à *false* pour toute autre configuration. C'est donc la valuation initiale des entrées de décision, se synchronisant avec l'état initial prévu dans la spécification du système S (ou S'). Et *atMostOne* est défini par :

$$atMostOne = \neg(\bigvee_{x=1}^{n-1}(d_x \wedge \bigvee_{y=x+1}^{n} d_y)) \tag{6.7}$$

Un seul $d_i \in \mathcal{D}$ au maximum peut être *true* afin que *atMostOne* soit également *true*.

Adaptation des transitions

Afin de finaliser un pas d'exécution, il est nécessaire d'adapter les transitions du LTS S'. En effet, bien que S et S' disposent du même espace d'état, leur transitions diffèrent car S' ne doit plus réagir que d'après l'ordre de reconfiguration donné par le système de décision (les inputs communs, $\mathcal{I}$, étant traités par S' dans les équations définissant $\mathcal{P}$ uniquement afin de calculer les prochaines configurations accessibles). Les transitions

de S' doivent également être adaptées au niveau de sorties, afin de produire (en plus des sorties communes $\mathcal{O}$) l'ensemble de booléens reflétant les configurations accessibles.

Soient S_k et S'_k (avec $0 \leq k < m$) respectivement un LTS de S et une transformation de S_k en tant que LTS de S'. Soit $\mathcal{B}_k$ un ensemble de variables booléennes et $n_k = card(\mathcal{B}_k) = card(\mathcal{Q}_k)$. La transformation de chaque $\mathcal{S}_k$ est donnée par :

$$\forall S_k = \langle \mathcal{Q}_k, q_{0_k}, \mathcal{I}_k, \mathcal{O}_k, \mathcal{T}_k \rangle, S'_k = \langle \mathcal{Q}_k, q_{0_k}, \mathcal{D}, \mathcal{O}'_k, \mathcal{T}'_k \rangle$$

avec $\mathcal{O}'_k = \mathcal{O}_k \cup \mathcal{B}_k$, $\mathcal{T}'_k \subset (\mathcal{Q} \xrightarrow{Bool(\mathcal{I})/\mathcal{O}'^*_k} \mathcal{Q})$, $\mathcal{O}'^*_k$ étant une conjonction de $\mathcal{O}'_k$, et $\mathcal{T}'_k$ étant défini par :

$$\begin{aligned} \mathcal{T}'_k = \{ (q_k & \xrightarrow{\{\bigvee_{i=1}^{n} d_i \in \mathcal{D} | q_i \in \mathcal{Q}, q'_k \in q_i\} / (a, \bigcup_{j=1}^{n_k} b_{q_{k_j}})} q'_k) | \\ & (q_k \xrightarrow{g/a} q'_k) \in \mathcal{T}_k \}, \\ b_{q_{k_j}} = & \begin{cases} true & \text{si } q_{k_j} = q'_k, q_{k_j} \in \mathcal{Q}_k \\ false & \text{sinon} \end{cases} \end{aligned} \tag{6.8}$$

Finalement, en accord avec le principe de composition synchrone, l'ensemble $\mathcal{T}'$ de $\mathcal{S}'$ correspond à la composition des transitions $\mathcal{T}'_k$:

$$\begin{aligned} \mathcal{T}' = & \{ (q \xrightarrow{(\bigwedge_{k=1}^{n} g_k)/(\bigwedge_{k=1}^{n} a_k)} q') | \\ & (q_k \xrightarrow{g_k/a_k} q'_k) \in \mathcal{T}'_k, (q_k, q'_k) \in q \times q' \} \end{aligned}$$

Lorsqu'une telle transformation est appliquée sur un LTS, chaque transition est physiquement conservée mais sa garde est complètement changée de manière à être *true* si et seulement si son état cible fait partie de la configuration donnée en entrée par le système de décision.

Nous venons ainsi de formuler les transformations nécessaires pour que S' – disposant du même espace d'états que S – respecte $\mathcal{F}$ étant donné $\mathcal{H}$ tout en garantissant à chaque pas les mêmes transitions potentielles d'après un même état et une même valuation de $\mathcal{I}$ dans S' et S. S' est donc équivalent à S d'après $\mathcal{F}$, $\mathcal{H}$, et $\mathcal{I}$, et est une implémentation acceptable de la spécification non-déterministe donnée en MARTE par le concepteur, tout en permettant à celui-ci d'implémenter et connecter son propre système de décision après contrôle.

6.4 Exemple de décision générique

L'objectif de cette section est de réexploiter une méthode générique de décision de reconfiguration, déjà appliquée dans le domaine des architectures reconfigurables [31], afin de montrer comment l'intégrer de manière contrôlée.

Dans le principe, cette méthode s'adresse principalement aux systèmes dynamiques dont les contraintes de reconfiguration reposent sur des critères antagonistes, typiquement : économie d'énergie, besoins en mémoire et autres critères de qualité de service. Elle est particulièrement intéressante dans les cas où l'évolution du système à piloter est difficilement prévisible, notamment parce que les critères et contraintes employés ne sont pas discrets mais quantitatifs (représenter en mémoire l'espace d'états correspondant à la discrétisation des évolutions possibles est souvent peu envisageable). L'approche consiste donc ici non pas à assurer formellement des contraintes temporelles pour le système à piloter, mais à optimiser ses reconfigurations, afin qu'il s'adapte au mieux à des contraintes dynamiques.

Après partitionnement du système à piloter en un ensemble de tâches, chacune possédant un ensemble d'implémentations possibles, la méthode est mise en œuvre en trois temps :

- les critères (qualité de service, etc.) de toutes les configurations – affectations d'une implémentation à chaque tâche – sont calculés statiquement, puis seules les configurations positionnées sur la frontière de Pareto de ses critères sont retenues (les autres ne seront jamais instanciées car considérées comme sous-optimales) ;
- les critères et contraintes étant quantitatifs, une technique issue du domaine du contrôle continu, la régulation proportionnelle et intégrale, est employée et ses coefficients doivent être calibrés ; l'idée étant d'intégrer l'erreur presque inévitable entre la consigne de reconfiguration (objectif de qualité à atteindre) et la qualité de la configuration effective (issu d'un ensemble fini) afin d'amortir les ordres de reconfiguration [32] ;
- afin qu'un choix puisse s'opérer parmi un ensemble fini de configurations d'après une consigne quantitative, un mécanisme de vote est mis en place dans la méthode ; dans notre cas, nous choisirons le *vote de Borda*, préconisée par les auteurs, dont nous verrons la technique plus loin.

À noter que cette méthode part du principe que toute configuration (de la frontière de Pareto conservée) est supposée correcte et donc instanciable à tout pas d'exécution. Une configuration est simplement plus ou moins optimale selon un objectif de qualité de service à atteindre. Si des configurations sont potentiellement incorrectes, les auteurs demandent de les écarter statiquement afin de ne jamais les instancier. Le type de système reconfigurable adressé est donc particulier, en effet une configuration *potentiellement incorrecte* (par rapport à une contrainte temporelle) n'est pas nécessairement incorrecte pour tout pas d'exécution ; l'écarter statiquement revient à sous-optimiser les possibilités de reconfigurations dynamiques. Il serait donc profitable à cette méthode d'être combinée

32. L'expression exacte employée par les auteurs étant *l'évitement des reconfigurations intempestives*.

avec notre approche de contrôle, afin de cibler les système pour lesquels il faut adapter les reconfigurations afin de garantir une exécution au sein d'un espace de configurations correctes recalculé à chaque pas de contrôle.

6.4.1 Frontière de Pareto dynamique

Le premier obstacle à résoudre dans cette intégration contrôle/décision, est celui de l'optimisation hors-ligne effectuée sur les configurations par l'approche de Pareto. La restriction de l'espace de configuration est en effet une étape réservée à la synthèse de contrôleur ; si cet espace est pré-réduit, il est possible que, pour un pas donné dans l'exploration des possibilités d'évolution, l'ensemble de configurations admises soit disjoint de l'espace pré-réduit, entraînant ainsi un échec de la synthèse de contrôleur dans notre méthode puisque l'une des contraintes temporelles à respecter est qu'à chaque pas, au moins une configuration doit être accessible.

L'optimisation via la méthode Pareto est intéressante, et il serait dommage de s'en priver car elle permet d'alléger le traitement de la décision en écartant par principe toute configuration dont l'ensemble des critères est optimisé par au moins une autre configuration. L'idée est ici d'exploiter cette méthode, mais en ligne. Dans notre approche, nous avons vu que les configurations sont organisées statiquement dans un treillis. Pour un pas donné de contrôle, lorsque l'espace des prochaines configurations accessibles est établi, il s'agit de combiner le treillis avec cet espace afin de calculer automatiquement la frontière de Pareto de l'espace, dans le but d'écarter les configurations accessibles, mais hors frontière, pour le processus de décision.

Nous proposons à cet effet un algorithme qui fonctionne en deux temps : la mise à plat statique (représentation vectorielle) du treillis afin de simplifier son exploration, et l'exploration en ligne de ce vecteur. L'algorithme 11 montre le principe de mise à plat d'un treillis dont $\top$ et $\bot$ sont respectivement la borne supérieure (root) et inférieure (leaf), en utilisant quatre opérations :

- *children* : $x \to E$, où x est une configuration, et E représente les *enfants* de x, c'est-à-dire l'ensemble des configurations directement inférieures à x dans le treillis ;
- *id* : $x \to i$, où x est une configuration, et i est l'identifiant de cette configuration, représenté par un entier ;
- *process* : x, marque la configuration x comme étant traitée ;
- *processed* : $x \to \mathbb{B}$, renvoie *true* si la configuration x a déjà été traitée.

Cet algorithme produit un vecteur d'entiers, nommé ici *vector*, composé sur le principe suivant : l'indice 0 contient le nombre n de configurations directement inférieures à *root*, si $n > 0$ l'indice 1 contient l'identifiant I de la première de ces configurations, l'indice 2 contient le nombre de configurations directement inférieures à la configuration I, si

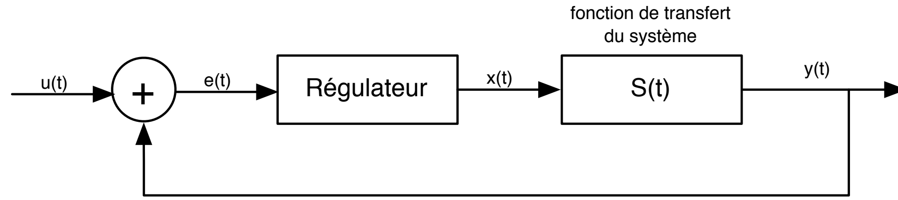


FIGURE 6.2 – Modèle en boucle fermée d'adaptation d'un système, avec un régulateur PI

ce nombre est supérieur à 0 alors l'indice 3 contient l'identifiant de la première de ces configurations, etc. Et ainsi de suite jusqu'à ce que toutes les configurations du treillis soient traitées.

L'algorithme 12 montre le principe d'exploration en ligne de ce vecteur, avec $v_x \in vector$, $p_y \in \mathcal{P}$, et avec l'aide des opérations suivantes :

- $tag : p_y$, servant à marquer la configuration p d'indice y ;
- $tagged : p_y \rightarrow \mathbb{B}$, renvoie *true* si la configuration p d'indice y est marquée ;
- $kill : p_y$, "tue" la configuration p d'indice k, c'est-à-dire qu'elle est marqué afin de ne plus être traitée, ainsi que ses enfants.
- $alive : p_y \rightarrow \mathbb{B}$, renvoie *true* si la configuration p d'indice y n'a pas été tuée ;

6.4.2 Régulation par contrôle PI

La logique de décision de reconfiguration employée par les auteurs repose sur un principe d'exécution en boucle fermée similaire à celui des systèmes réactifs, tel que décrit dans figure 6.2, où $u(t)$ est la consigne utilisateur, $y(t)$ est la grandeur contrôlée du système, $e(t)$ est l'erreur calculée entre $u(t)$ et $y(t)$, et $x(t)$ est la traduction régulée de la consigne interprétée par le système.

Dans un système reconfigurable idéal, un espace infini de configurations serait disponible à chaque pas, et pour toute consigne comportementale imposée en ligne par l'utilisateur, il existerait une configuration appropriée n'ayant aucune erreur vis-à-vis de la consigne. Pour un tel système hypothétique, la solution de contrôle continu suivante peut être mise en œuvre (avec k_p en tant que coefficient proportionnel du régulateur) :

$$x(t) = k_p e(t) \tag{6.9}$$

Et y(t) étant la grandeur contrôlée observée en sortie du système après adaptation, cette fonction introduit un effet d'intégration qui peut, en théorie, annuler l'erreur statique tel que :

$$y(t+1) = y(t) + x(t) \tag{6.10}$$

```
1  begin
2      vector ← ∅ ;
3      E ← children(⊤);
4      N ← ∅ ;
5      while E ≠ ∅ do
6          vector ← vector + {card(E)};
7          foreach {i | 0 ≤ i < card(E)} do
8              vector ← vector + {id(e_i)} ; // e_i ∈ E
9              if {e' ∈ children(e_i) | e' ≠ ⊥} then
10                 vector ← vector + {card(children(e_i))};
11                 foreach {j | 0 ≤ j < card(children(e_i))} do
12                     vector ← vector + e'_j ; // e'_j ∈ children(e_i)
13                     if processed(e'_j) = false then
14                         N ← N + e'_j;
15                         process(e'_j);
16                     end
17                 end
18             else
19                 vector ← vector + 0;
20             end
21         end
22         E ← N ;
23         N ← ∅;
24     end
25 end
```

Algorithme 11: Représentation vectorielle du treillis de configurations

```
1  begin
2      ok ← false;
3      i ← 0;
4      while (v_i ≠ 0 ∧ ¬ok) do
5          ok ← true;
6          i ← i + 1;
7          foreach {j | 0 ≤ j < v_i} do
8              if tagged(p_{v_i}) then
9                  foreach {k | 0 ≤ k < v_{i+1}} do
10                     tag(p_{v_{i+1+k}}) ; /* on marque les enfants de p_{v_i} car une
                       meilleure configuration qu'eux est potentielle */
11                 end
12                 kill(p_{v_i}) ; /* on supprime p_{v_i} afin de ne pas marquer une
                   nouvelle fois ses enfants */
13             else if alive(p_{v_i}) then
14                 if p_{v_i} = 0 then
15                     ok ← false; /* p_{v_i} n'est pas une configuration potentielle, il
                       faudra traiter ses enfants aux prochaines itérations */
16                 else if p_{v_i} = 1 then
17                     foreach {k | 0 ≤ k < v_{i+1}} do
18                         tag(p_{v_{i+1+k}}) ; /* p_{v_i} est une configuration potentielle, ses
                           enfants ne peuvent donc pas être retenus car moins
                           optimaux */
19                     end
20                     pareto ← pareto + v_i /* enregistrement de l'indice de cette
                       configuration potentielle */
21                 end
22             end
23             i ← i + v_{i+1} + 2 ; /* indice (dans le vecteur) du prochain enfant de la
               configuration courante s'il en reste à traiter, ou indice pour la taille
               de l'ensemble des enfants la prochaine configuration traitée */
24         end
25     end
26 end
```

Algorithme 12: Construction dynamique de la frontière de pareto d'un ensemble de configurations potentielles

Dans la réalité, un système reconfigurable, tel que ceux que nous considérons dans notre approche, dispose d'un nombre limité de configurations. Le modèle classique PI a donc été adapté par les auteurs afin de prendre en compte trois caractéristiques :

- l'objectif n'est pas d'annuler l'erreur statique, puisque c'est généralement impossible, mais de la minimiser ;
- les reconfigurations sont coûteuses, il est nécessaire de stabiliser les reconfigurations afin d'éviter les changements intempestifs si le non-respect de la consigne est mineur ;
- le traitement s'effectue en ligne, la solution doit être simple en terme de temps et de ressources nécessaires à son exécution.

Ces caractéristiques sont compatibles avec notre positionnement de contrôle : 1) les éventuelles erreurs de décision (erreur statiques) ne sont pas une menace pour le respect des objectifs de contrôle, en effet nos contrôleurs fournissent un espace de configurations accessibles, donc toutes acceptables d'après un objectif de contrôle, et ce, quelle que soit leur optimalité ; 2) la gestion des reconfigurations intempestives est un plus par rapport à notre approche, en effet nous ne régulons pas cet aspect qui est du ressort du contrôle continu ; 3) nous sommes également dans un contexte embarqué, dont les ressources sont potentiellement très limitées et dont les contraintes temps réel peuvent être fortes, un algorithme de décision peu coûteux est compatible avec ce que nous recherchons.

Les auteurs du système de décision proposent d'adapter en conséquence le modèle de régulation proportionnel et intégral par la définition suivante, où k_i est le coefficient intégral :

$$x(t) = k_p e(t) + k_i x(t-1) \tag{6.11}$$

Le choix de la configuration d'indice j à l'instant $t+1$ s'effectue alors par :

$$y_j(t+1) = max(\{y_k(t+1) \mid y_k(t+1) \leq u(t) + x(t)\}) \tag{6.12}$$

k étant dans notre cas compris entre 0 et card($\mathcal{P}$), avec pour contrainte de ne considérer que les cas où $p_k = true, p_k \in \mathcal{P}$, contrainte surmontable par le fait que l'on ne transmet (après contrôle) au processus de décision qu'un ensemble de configurations nécessairement correctes.

Dans le cas idéal, avec ce principe de régulation on obtiendrait $y(t+1) = y(t)+x(t)$, mais puisque l'espace de configuration n'est pas infini, les auteurs considèrent une différence, modélisée en tant que perturbation aléatoire (notée $\epsilon(t)$), entre cette sortie idéale supposée (nommée $\tilde{y}(t)$) et la sortie observée $y(t)$, tel que :

$$y(t) = \tilde{y} + \epsilon(t) \tag{6.13}$$

Le lecteur intéressé par l'implémentation effective de cette méthode de décision pourra trouver dans [31] les détails techniques supplémentaires concernant le contrôle de systèmes

perturbés ainsi que les méthodes de tuning des coefficients (proportionnel et intégral) tout en gardant un mécanisme de régulation stable.

En ce qui nous concerne dans cette étude, à des fins d'intégration, seule la consigne $u(t)$ manque encore à notre modélisation. Un emplacement idéal pour sa spécification est en tant que port du RtUnit de contrôle faisant usage de la décision, lié en interne vers le *resolver* de ce contrôleur. Un tel port a été montré à titre d'exemple lors de la présentation du *resolver* dans MARTE : on modélise bien ainsi dans MARTE ce principe contrôle/décision en un même RtUnit centralisant les événements en provenance de l'environnement (utilisateur, capteurs, etc.) dont la consigne fournie à la décision fait partie. L'ambiguïté est également levée entre les types d'entrées via cette représentation : les entrées non connectées au *resolver* relèvent ainsi du contrôle pur (donc réservés à la transformation vers BZR), tandis que les entrées connectées au *resolver* relèvent de la décision (employées par le concepteur à la réalisation du mécanisme de décision).

Une autre contrainte s'impose, cette fois au niveau du principe d'exécution : la décision présentée ici est pilotée par le temps, il n'est donc pas possible de déclencher arbitrairement des pas de contrôle sans risquer de fausser la décision. Un tel mécanisme contrôle/décision doit alors s'adapter à ce prérequis : les événements de contrôle et de décision doivent être synchronisés sur une même horloge régulière dans le temps.

6.4.3 Élection d'une configuration

Après avoir régulé la consigne utilisateur, il doit être possible, pour une consigne donnée, de choisir une configuration appropriée. La sélection d'une configuration, donnée par l'équation 6.12 repose sur une fonction nommée *max* non détaillée jusqu'ici. On comprend qu'il s'agit de trouver la configuration (accessible) la plus proche possible de la consigne, mais lorsque les configurations sont évaluées sur plus de critères, souvent antagonistes, la méthode de sélection (donc l'implémentation de *max*) n'est pas triviale.

Toujours dans le même respect des contraintes de légèreté algorithmique (temps, ressources, etc.) et de généricité de l'approche, les auteurs évaluent plusieurs algorithmes, et retiennent au final la méthode du vote de Borda. Il s'agit d'un vote prenant en compte l'ordre des candidats (ici, l'ordre des configurations). Pour chaque votant (ici, chaque critère muni de sa consigne à atteindre), les candidats sont rangés par *ordre de mérite*. Un poids est attribué à chaque candidat en fonction de son rang. Le poids n est assigné au candidat préféré pour un vote comportant n candidats. Ce poids est ensuite décrémenté d'une unité pour chaque candidat successif. Enfin, la somme des poids pour chaque candidat et chaque votant est calculée. Le candidat obtenant le meilleur score est alors élu. Dans la pratique, si plusieurs candidats obtiennent le meilleur score, on choisira un can-

didat arbitrairement (par exemple en donnant une priorité sur les poids, puis si ce n'est pas toujours pas suffisant, en sélectionnant le candidat dont l'indice est le plus petit).

Cette fois-ci, aucune contrainte particulière n'est relevée dans le cadre de l'intégration du vote de Borda avec notre mécanisme de contrôle. Les configurations pilotées par un contrôleur sont simplement ordonnées statiquement en autant de vecteurs qu'il y a de poids déclarés pour ce contrôleur. Il suffit ensuite pour le concepteur d'implémenter le mécanisme de vote tel qu'il vient d'être défini afin de l'exécuter en ligne pour sélectionner à chaque pas une configuration proche d'une consigne arbitraire.

6.5 Conclusion

Le présent chapitre a montré le principe d'intégration contrôle/décision, permettant ainsi de séparer – pour un système reconfigurable donné – ce qui relève purement de la sécurité de reconfiguration (et qui doit être traité par une approche logique discrète), de ce qui relève de l'optimisation.

La méthode ainsi proposée a permis d'exposer le mode d'interfaçage générique pour l'implémentation arbitraire d'un module de décision : il s'agit, pour ce module, de prendre un vecteur de booléens en entrée, chaque booléen définissant l'accessibilité ou non d'une configuration, et de produire en sortie un vecteur de même taille, reposant sur le même principe mais cette fois-ci avec un seul booléen à *true* et avec pour contrainte le fait que la configuration correspondante de ce booléen doit être accessible (donc son booléen d'entrée correspondant est *true*).

Il a ensuite été vu comment intégrer cette méthode avec un mécanisme de décision déjà éprouvé. Le mécanisme de régulation proportionnelle et intégrale est en effet intéressant en raison de sa généralité. L'intégration de ce mécanisme, en exploitant les résultats obtenus dans [31], a permis de montrer plusieurs points intéressants valables pour l'intégration de tout système de décision :

- les éventuelles optimisations hors ligne de l'espace de configuration doivent être soit éliminées, soit réimplémentées pour une exécution en ligne : l'espace statique doit être le plus complet possible afin de favoriser une réussite de la synthèse de contrôleur qui elle, est capable de déterminer formellement quelle configuration peut être conservée (et ce, sur des critères dynamiques) ;
- les événements dédiés à la décision seule doivent être identifiables par rapport aux autres événements dans la spécification de contrôle en MARTE, en effet les événements de décision relèvent de la responsabilité du concepteur et ne doivent pas être gérés dans la représentation synchrone du contrôleur, qui ne traite que de l'aspect contrôle et non de la décision ;

– le modèle de calcul tel qu'il a été défini pour l'exécution du contrôle impose aux événements de l'interface d'un contrôleur (RtUnit stéréotypé «controller») d'être synchronisés ; les éventuelles contraintes d'exécution, soit du contrôle, soit de la décision, doivent s'accorder ; dans l'exemple de décision présenté, les contraintes temps réel de l'exécution de la décision se reportent sur l'exécution du contrôle (qui à la base, n'est pas piloté par le temps) : donc soit les événements de contrôle arrivent de manière régulière en même temps que les événements de décision, soit il faut adapter la spécification de contrôle afin qu'au déclenchement d'un pas de contrôle, l'état interne du contrôleur puisse être figé si ce n'est pas le moment approprié pour réagir (un booléen en entrée de son interface peut par exemple servir à inhiber toutes ses transitions quelles que soient les valeurs des autre entrées, cependant il devra émettre un ensemble de configurations à chaque appel, conformément au modèle de calcul).

Ce chapitre clôt la partie théorique de cette étude. La prochaine partie concernera la mise en pratique de cette méthodologie de modélisation, de transformation, de synthèse de contrôleur, et de décision.

Troisième partie

Validation, développement et évaluation

7

Définition d'une architecture de démonstration

Sommaire

7.1 Introduction **145**
7.2 Définition des besoins **146**
7.2.1 Capacité de reconfiguration 147
7.2.2 Méthode d'exécution 147
7.3 Spécifications de l'architecture cible **148**
7.3.1 Établissement d'un pipeline générique de traitement vidéo . 149
7.3.2 Adaptativité du pipeline 150
7.4 Spécification du type d'application ciblée **152**
7.4.1 Démonstrateur pour la synthèse de contrôleur 152
7.4.2 Application de suivi d'objets 153
7.5 Matériel considéré **154**
7.6 Conclusion **154**

7.1 Introduction

La mise en œuvre concrète des théories développées dans FAMOUS est une tâche majeure, partagée par l'ensemble des acteurs du projet. Dans le cadre de cette étude, il s'agit donc de cerner les besoins qui contribueront à déterminer une plateforme globale de démonstration, grâce à laquelle il sera possible de constater l'efficacité et l'applicabilité des outils et méthodes avancés.

L'objectif de ce chapitre est dans un premier temps de synthétiser les prérequis minimaux de cette plateforme. Il s'agit en quelque sorte de proposer un cahier des charges de

la plateforme – avec une focalisation sur les propositions de cette étude – en récapitulant ce qui a été théorisé jusqu'ici en vue de le mettre en œuvre. Dans un deuxième temps, le côté architectural de la plateforme est défini et justifié en fonction des prérequis évoqués. L'enjeu est d'établir les éléments clés de l'architecture, à partir desquels il sera possible de définir un large panel d'applications intéressantes pour illustrer le projet. Dans un troisième temps, le type principal d'application ciblée est plus particulièrement détaillé. Le but est de préparer l'implémentation d'un exemple approprié. Enfin, le choix du matériel capable d'instancier la plateforme est expliqué.

La présente étude se terminant en avance de phase dans le projet FAMOUS, l'objectif global de la partie "validation" de ces travaux est de réaliser un première proposition de plateforme, de la valider par l'exemple, et d'expliquer les hypothèses selon lesquelles les autres acteurs du projet peuvent s'intégrer et passer à l'échelle en incorporant leurs propres avancées/fonctionnalités.

Cette proposition reprend notamment les idées clés que nous avons publiées dans [44], en terme de spécification de plateforme reconfigurable définissable dans une méthodologie de modélisation.

7.2 Définition des besoins

Ce qui vient d'être défini dans cette étude est qualifiable de *flot de conception*, il s'agit d'un socle à la fois autonome et modulaire. *Autonome* par le fait qu'il doit se suffire à lui-même dans le cadre d'une mise en œuvre d'un exemple complet : depuis une spécification de contrôle en MARTE, jusqu'à l'exécution sous réserve que les aspects architecturaux et applicatifs (implémentations) soient préparés. Et *modulaire*, car les divers acteurs du projet FAMOUS viendront greffer leurs modules sur ce socle.

Le module défendu dans cette étude est celui de l'automatisation du contrôle, et sera donc ici l'objet principal de démonstration. Il s'agira également de prendre en compte les autres modules venant compléter ce flot : une spécification complète de l'architecture et des applications supportées par la plateforme sera proposée dans un module dédié à la modélisation, l'interopérabilité de la spécification sera plus particulièrement traitée par un module dédié à IP-XACT, un module consacré à la diffusion d'un ordre de configuration dans un graphe de tâches viendra augmenter les possibilités d'exécutions, et enfin, un module essentiellement orienté *contrôle* viendra compléter le module même de cette étude afin de permettre à la fois plus de possibilités de contrôle – en particulier au niveau de l'ordonnancement (tâche après tâche) – ainsi qu'une optimalité garantie.

7.2.1 Capacité de reconfiguration

Le premier besoin, presque transversal à tout module, est la capacité de la plateforme proposée à permettre le développement de systèmes dynamiquement reconfigurables. L'idéal étant la possibilité de mettre en œuvre visuellement la notion de reconfiguration dynamique et partielle.

Pour rappel, les éléments MARTE «Configurations» mentionnés dans cette étude sont employés pour la définition partielle des configurations du système : lorsque le ou les mode(s) d'une Configuration MARTE sont actifs, la spécification donnée dans ce composant – mappages, valuations de propriétés, etc. – doit être instanciée. En conformité avec la capacité de modélisation des Configurations MARTE, la plateforme proposée doit permettre d'illustrer les reconfigurations matérielles suivantes :

- la modification de co-processeurs : le jeu d'instruction d'un processeur classique – contenu ou instancié dans la plateforme – doit pouvoir être étendu au moyen d'ajouts/modifications dynamiques de coprocesseurs arithmétiques (cette possibilité, à prévoir dans FAMOUS, ne sera pas illustrée dans cette étude car la modélisation d'architectures reconfigurables dans MARTE n'est pas assez mature à ce stade du projet) ;
- la modification d'accélérateurs sur un bus : il s'agit ici concrétiser l'allocation d'un composant du modèle d'application (une tâche) vers un composant du modèle d'architecture (une *black box* définissant l'interface à respecter pour toute instance) ; toute implémentation de la tâche mentionnée définit une implémentation matérielle possible (c'est-à-dire un bitstream partiel) pour cette black box.

7.2.2 Méthode d'exécution

La méthode générale d'exécution des systèmes prévus pour cette plateforme doit reposer sur deux critères importants à illustrer dans cette étude : l'existence d'une boucle de contrôle globale et la définition d'un modèle de calcul.

Boucle de contrôle

Conformément à la définition générique des systèmes réactifs, l'exécution des systèmes considérés ici doit pouvoir s'assimiler à une boucle infinie dans laquelle, à chaque itération, des événements sont captés depuis l'environnement, puis traités par le système qui produit des sorties. Ce socle commun d'exécution doit permettre d'insérer les éventuels contrôleurs donnés dans la spécification MARTE : chaque itération de la boucle d'exécution déclenche un pas de contrôle dont le rôle est d'analyser des événements d'entrée afin de prendre une décision de reconfiguration.

Modèle de calcul

Le mécanisme de diffusion des ordres de reconfiguration (la *membrane*) n'étant pas encore pleinement intégré dans FAMOUS, il s'agit ici de préparer cette éventualité en proposant une plateforme dans laquelle il est aisé d'ajuster un modèle de calcul arbitraire, régissant la manière d'exécuter les tâches spécifiées.

Le principe de séparation contrôle/données mis en avant dans cette étude a en effet pour but de rendre indépendants les mécanismes de contrôles et ceux dédiés à la gestion (exécution, suspension, reprise, etc.) des tâches, ne contraignant ainsi pas l'implémentation d'un modèle de calcul arbitraire.

La membrane aura pour enjeu de les synchroniser correctement, il s'agira alors d'ajuster la cadence de contrôle par rapport à celle imposée par le modèle de calcul (le début d'une séquence de tâches dans un tel modèle doit déclencher un pas de contrôle). En l'absence de membrane pour le moment, il sera proposé un modèle de calcul simple, basé sur les réseaux de Kahn, dans le cadre du démonstrateur de cette étude.

7.3 Spécifications de l'architecture cible

Nous allons ici repartir de la plateforme à base de FPGA Xilinx présentée dans [16], dont les caractéristiques principales sont de supporter l'acquisition d'un flux video, et de déclencher des reconfigurations dynamiques à partir de bitstreams partiels accessibles en réseau.

Le choix de cette plateforme permet ici combler deux étapes non finalisées à ce stade du projet :

- l'absence de mécanismes de modélisation appropriés pour la représentation d'architectures reconfigurables dans MARTE ; l'idée est ici de pouvoir nous focaliser essentiellement sur les modèles d'application et d'allocation présentés dans cette étude ;
- l'absence de membrane : une fonction *loadbitstream*, présente de base dans la partie applicative de la plateforme a pour rôle d'instancier un bitstream partiel à partir de son identifiant ; en se basant sur une spécification d'allocations simples telle que *une Configuration MARTE active = un bitstream partiel à charger*, il s'agit ici de connecter immédiatement la sortie du contrôleur – constitués des identifiants de bitstreams formant une configuration – à des appels de cette fonction afin d'instancier à chaque pas de contrôle la configuration adéquate du système ; nous nous restreignons ainsi à un cas simple et générique d'exécution, afin de nous dispenser de la membrane : les éventuelles allocations plus particulières ne seront pas traitées ici car il relève du rôle essentiel de la membrane de savoir comment les instancier.

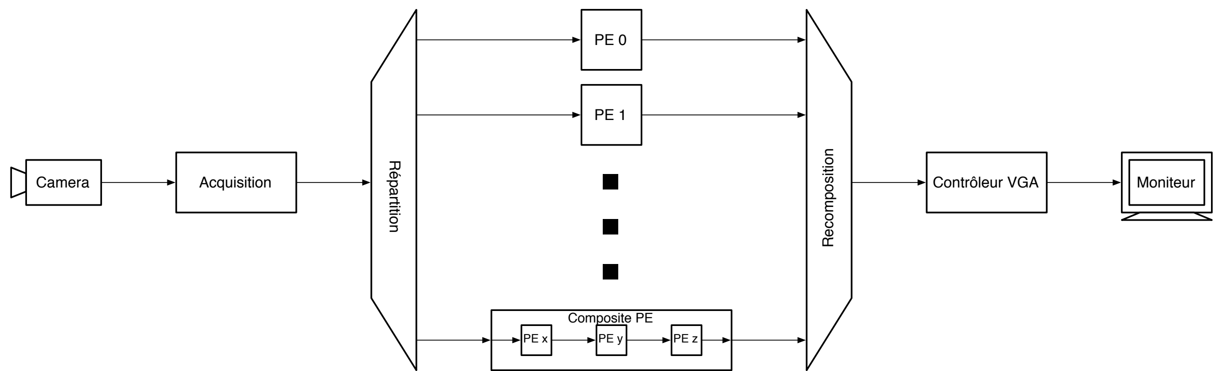


FIGURE 7.1 – Pipeline de traitement vidéo

À terme, il s'agira bien entendu de modéliser complètement la plateforme, avec une membrane appropriée. Mais pour les besoins précis de cette étude, l'objectif est maintenant de spécialiser cette plateforme afin de pouvoir y déployer des systèmes reconfigurables – sous contrôle – dédiés au traitement de flux vidéo.

7.3.1 Établissement d'un pipeline générique de traitement vidéo

Il s'agit ici de construire un framework en tant que modèle d'architecture pour déployer une application donnée. Nous allons ainsi focaliser le champ de modélisation/conception sur les concepts présentés dans cette étude, en admettant que ce qui est implémenté manuellement et statiquement dans la plateforme sera à terme pris en charge (génération de code) par les autres modules du projet FAMOUS.

Le choix d'exploiter le flot d'acquisition vidéo supporté par cette plateforme se justifie par le fait que nous recherchons un moyen visuel et didactique de présenter les travaux. L'établissement d'un framework pour le déploiement de systèmes de traitement vidéo est donc légitime.

La capture et l'affichage de la vidéo sort du cadre de cette étude, elle est donc implémentée statiquement et fournie par le framework en tant que brique logicielle/matérielle pour le support de conception et déploiement. Tout d'abord, les différents traitements du flux vidéo ou *Processing Elements* (PE) – dont l'implémentation en tant que softcores est à la charge du concepteur – sont organisés en réseau de manière à pouvoir être composés en séquence et en parallèle, afin de supporter l'équivalent d'un graphe de tâches.

La figure 7.1 montre le pipeline de traitement vidéo proposé. Techniquement, un certain nombre de PE sont alimentés en terme de pixels par une tâche (statiquement implémentée) nommée *répartition* elle-même alimentée en amont par des trames VGA provenant d'une tâche *acquisition* traduisant le flux vidéo issu d'une caméra. Chaque PE travaille ainsi sur une zone d'une image, et peut être spécifié de manière composite : le

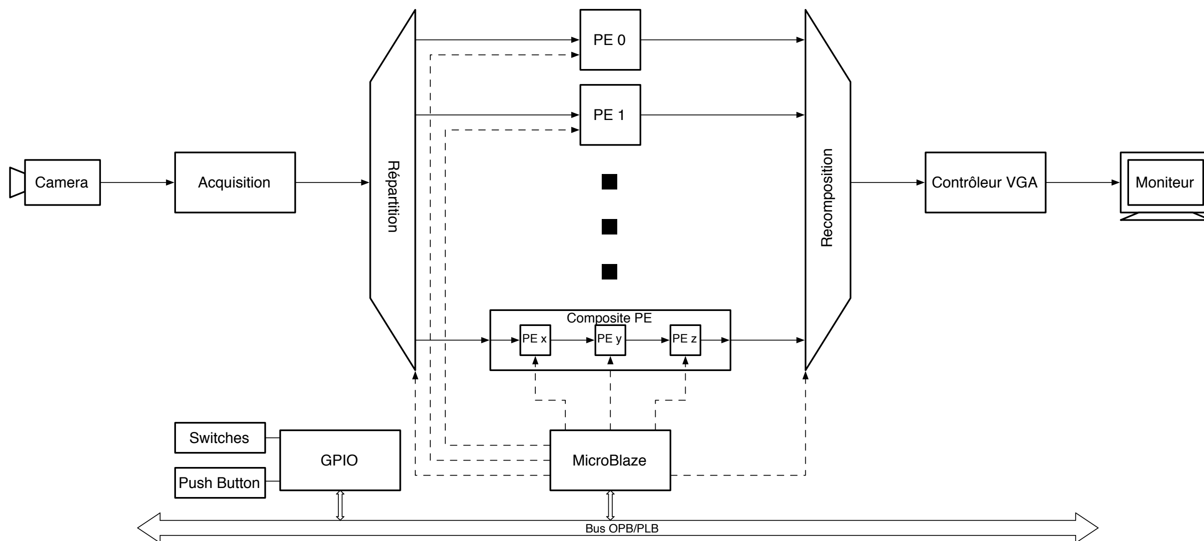


FIGURE 7.2 – Contrôle bas-niveau de la reconfiguration, effectué via un MicroBlaze

traitement effectué par un PE composite est défini par l'exécution en séquence des PE qu'il encapsule. En fin de traitement d'une image donnée, une tâche *recomposition* agrège les résultats de traitements des PE pour chaque zone de la répartition et envoie l'image ainsi filtrée au contrôleur VGA qui la restitue vers un moniteur.

La cadence d'exécution imposée par ce framework est assez implicite : une séquence de tâches mappées sur une telle architecture s'exécute sur une image complète du flux vidéo. Afin de synchroniser correctement – pour la suite – le traitement du contrôle (ordres de reconfiguration) et des données (PE), la partie contrôle ne devra donc pas excéder un ordre de reconfiguration par image.

7.3.2 Adaptativité du pipeline

Reconfiguration dynamique et partielle

L'objectif étant de montrer une certaine forme de dynamicité, chaque PE (softcore) est supposé reconfigurable indépendamment [33] des autres, illustrant ainsi la capacité de reconfiguration dynamique et partielle de la plateforme. En cas de réception d'un ordre de reconfiguration, le principe est d'employer le dispositif – implémenté de base – chargé de récupérer les bitstreams partiels correspondants auprès du serveur de bitstreams puis d'exploiter un processeur Microblaze afin que celui-ci les mette en place sur le FGPA conformément à l'approche classique (pour Xilinx) de reconfiguration dynamique.

La figure 7.2 montre (en flèches pointillées) les possibilités de pilotages issues du Mi-

33. En l'absence de contraintes temporelles pour le moment

croBlaze. On notera la possibilité de piloter (reconfigurer) les tâches *répartition* et *recomposition*, bien que cette particularité ne sera pas démontrée dans cette étude : à terme l'ensemble de l'application devra être spécifié en MARTE, donc ces deux tâches apparaitront en tant que tâches du modèle (RtUnits) et seront à ce titre potentiellement reconfigurables au lieu d'être des éléments statiques fournis par le framework. On note également le support de certains ports *General Purpose Input/Ouput* (GPIO), notamment des switches et boutons afin que l'utilisateur puisse produire des événements manuellement.

Paramétrisation des éléments du pipeline

Autre possibilité de modification intéressante : les PE tels qu'ils sont définis de manière générique dans la plateforme sont paramétrables. Un programme, s'exécutant sur le MicroBlaze, peut ainsi adapter dynamiquement les paramètres de positionnement et de dimensions des parcelles d'images traités par les PE. Comme nous l'avons précisé, la modification de paramètres de tâches est représentable dans MARTE, via les éléments *Configuration*. Mais il s'agit ici d'un mécanisme différent de la reconfiguration matérielle telle que nous souhaitons modéliser dans cette étude en restreignant les modalités d'utilisation des Configurations MARTE aux allocations simples. D'un point de vue *contrôle*, après l'émission d'un ordre de configuration (caractérisé par les identifiants des modes actifs), il faudrait analyser les Configurations MARTE activées afin de traiter au cas par cas la manière de les instancier (reconfiguration matérielle ou changement de paramètre), ce qui devient le rôle de la membrane. Or nous souhaitons ici garder un rôle simple de membrane – car elle sera implémentée manuellement et statiquement en tant qu'élément du framework – en ne permettant que les reconfigurations matérielles des PE.

Adaptation purement matérielle

De même, lorsque les mécanismes de modélisation d'architectures reconfigurables seront effectifs dans MARTE (c'est-à-dire dans RecoMARTE), il devra être permis d'illustrer via cette plateforme des mises en œuvre de reconfiguration dynamique impactant non plus des tâches (PE, répartition, etc.), mais des éléments relevant purement de l'architecture (par exemple des co-processeurs).

Afin de préparer cette éventualité, nous allons augmenter le framework afin de simplifier la mise en œuvre d'algorithmes (reconfigurables) d'analyse d'image. Pour cela, nous proposons d'employer un PE particulier, *collecteur* dans la figure 7.3, afin de stocker une image complète en mémoire, afin que celle-ci soit traitée par une autre tâche[34]. Ce type de traitement s'effectuerait complètement en parallèle des autres PE, à un fréquence

34. Par exemple pour la reconnaissance de formes, le suivi d'objets, etc.

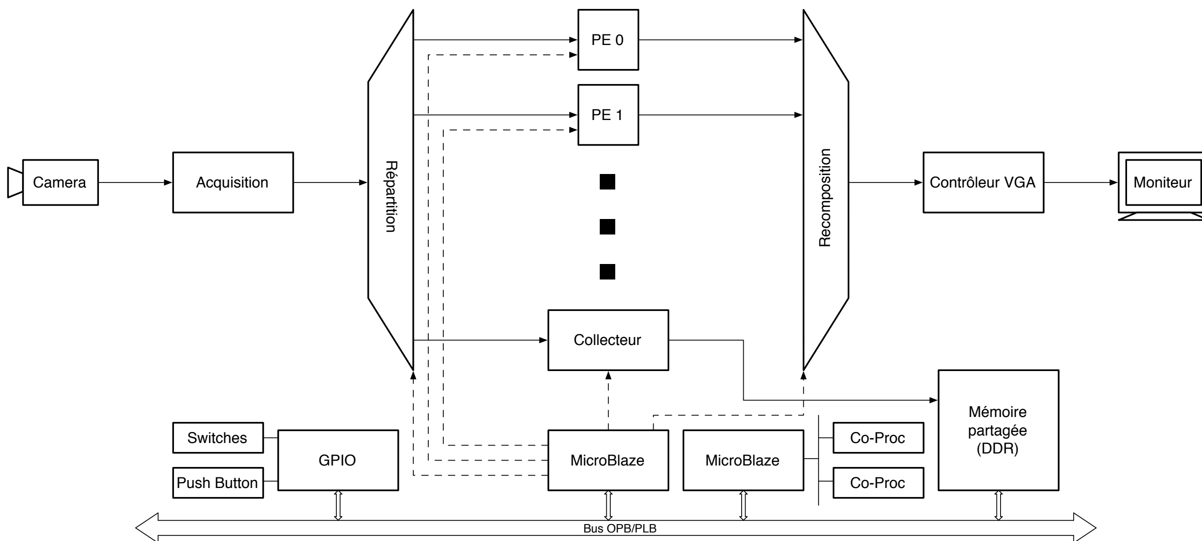


FIGURE 7.3 – Ajout de capacités de traitements d'images exploitant des co-processeurs

indépendante. Selon la cadence recherchée pour ce traitement, et en fonction de sa complexité, il pourra être motivant d'exploiter des co-processeurs à des fins d'accélération, en l'occurrence des co-processeurs reconfigurables.

7.4 Spécification du type d'application ciblée

Le modèle d'architecture ainsi établi va permettre de déployer rapidement des applications à la fois dynamiques et orientées *traitement d'images*. Cette section expose dans un premier temps une perspective d'utilisation de la synthèse de contrôleur minimaliste, permettant de montrer toutes les étapes de conception/déploiement, sans être surchargé par l'explosion combinatoire se produisant lors des transformations. Puis dans un deuxième temps nous verrons une perspective plus réaliste d'utilisation, dans laquelle les différents acteurs du projet FAMOUS pourront mettre en œuvre leurs modules respectifs.

7.4.1 Démonstrateur pour la synthèse de contrôleur

Conformément à l'approche de contrôle présentée dans cette étude, l'évolution des systèmes reconfigurables ciblés devrait pouvoir dépendre de facteurs externes non nécessairement prédictibles ou cadencés régulièrement.

L'architecture intermédiaire, donnée en figure 7.2, suffira amplement pour supporter un tel critère : une application composée de tâches reconfigurables traitant un flux vidéo pourra être déployée avec en son sein un flot de contrôle captant des événements produits par l'utilisateur à une cadence arbitraire via les GPIO. L'idée est que l'utilisateur puisse

être responsable du déclenchement des pas de contrôle, indépendamment du flot de données (sous réserve que deux pas de contrôle concernent deux images distinctes dans le flux vidéo, conformément à la contrainte de synchronisation contrôle/données). Les éventuels changements d'implémentations des tâches de l'application s'effectueront d'après des contraintes données en amont par le concepteur dans la spécification de contrôle en MARTE. Il sera intéressant de pouvoir illustrer deux types de comportement du contrôleur synthétisé d'après ces contraintes :

- l'interdiction d'effectuer une transition ;
- le forçage de transition, avec notamment plusieurs transitions proposées, afin qu'un algorithe de décision puisse effectuer un choix.

Cette perspective d'utilisation sera mise en œuvre au prochain chapitre, où un exemple d'application modélisé en MARTE – sous la forme d'un graphe de tâche muni d'une spécification de contrôle – sera transformé, synthétisé et exécuté.

7.4.2 Application de suivi d'objets

À des fins de démonstrations plus avancées, la mise en œuvre d'une application de traitement d'images plus réaliste sera préférable pour les autres acteurs, notamment dans le but d'illustrer la modélisation complète de l'application (graphe de tâches paramétrable, membrane, etc.) et de l'architecture (co-processeurs reconfigurables, etc.).

L'implémentation d'une application de suivi d'objets serait par exemple une instance intéressante de ce type de systèmes réalistes.

Un tel exemple, dépassant la portée de cette étude, mérite néanmoins d'être conceptualisé ici. Il s'agit d'exploiter l'architecture complète, donnée en figure 7.3, dans le but de montrer le flot FAMOUS complet lorsqu'il sera prêt et d'y déployer une application faisant usage de la reconfiguration dynamique et partielle. Le principe repose sur l'analyse d'images en provenance du flux vidéo par une tâche – implémentant un algorithme de détection et de suivi d'objet – exécutée sur un MicroBlaze et pouvant tirer parti d'un ou plusieurs co-processeurs reconfigurables pour accélérer son traitement. Lorsqu'aucun objet n'est détecté, l'image complète peut-être traitée par un PE, laissant l'image intouchée. Cela correspond à un mode d'attente (wait). Dès qu'un objet est détecté, le système doit se configurer en mode suivi (tracking) : l'image est divisée en 5 zones, chacune traitée par un PE dont la configuration par défaut est la suivante :

- PE 1 : désigne l'espace au-dessus de l'objet suivi, affiché en niveau de gris ;
- PE 2 : désigne l'espace à gauche de l'objet suivi, affiché en niveau de gris ;
- PE 3 : désigne l'espace de l'objet suivi, affiché en couleur ;
- PE 4 : désigne l'espace à droite de l'objet suivi, affiché en niveau de gris ;
- PE 5 : désigne l'espace au-dessous de l'objet suivi, affiché en niveau de gris ;

Avec une telle proposition, il devient possible de mettre en œuvre les variations (donc les possibilités de reconfigurations) suivantes :

– l'alternance entre plusieurs algorithmes de suivi, dépendant de critères tels que les conditions lumineuses, le niveau de batterie, etc. ;
– l'adaptation des tailles des tailles des espaces de suivi ;
– la modification du nombre d'espaces de suivi (gestion de plusieurs objets par exemple) ;
– le changement dynamique des traitements effectués par les PE (couleur ou niveau de gris).

7.5 Matériel considéré

Les deux contraintes principales dans le choix du matériel pour la réalisation du démonstrateur sont la présence 1) d'un FPGA dynamiquement reconfigurable et 2) des entrées/sorties vidéo. Le choix s'est porté sur deux cartes de développement Xilinx : une XUP-V5, basée sur un FPGA Virtex 5, et une ML-605, basée sur un FPGA Virtex 6.

Le choix de la XUP-V5 vient du fait qu'elle est particulièrement bien maîtrisée parmi les membres du projet FAMOUS dont certains ont participé directement ou indirectement aux projets MoPCoM et OPEN-People[35], faisant également usage de cette carte. Le démonstrateur a donc été en testé en priorité sur la XUP-V5, avec un traitement vidéo limité en résolution à 640x480.

Puis, afin de moderniser le matériel, mais également dans le but de montrer la portabilité du flot, la ML605 a été choisie, puis augmentée d'une carte de traitement vidéo rendant ainsi le tout capable de traiter un flux vidéo en haute définition.

Les éventuels conflits de portabilité concernent essentiellement la spécification matérielle, et n'impactent pas la spécification de contrôle et ses transformations associées (purement logicielles). La question de portabilité dépasse donc le cadre de cette étude, c'est pourquoi nous effectuerons ici nos déploiements immédiatement sur la ML605.

7.6 Conclusion

Ce chapitre a montré les différentes étapes de conceptualisation d'une plateforme adaptée à la mise en œuvre d'un démonstrateur pour le projet FAMOUS.

Dans le cadre de cette étude, nous nous limiterons au strict nécessaire afin de créer une preuve de concept. L'architecture intermédiaire, proposée en figure 7.2, a été justifiée à cet effet pour sa capacité à supporter un graphe de tâches reconfigurable, allié à un flot de contrôle maîtrisable arbitrairement par l'utilisateur.

35. `https://www.open-people.fr/bin/view/Main/`

L'architecture complète qui sera employée à terme dans le projet FAMOUS n'est pas indispensable ici, et nous utiliseront un sous-ensemble pour déployer le démonstrateur de cette étude. Concernant le choix matériel, la deuxième plateforme à avoir été intégrée au projet – la ML605 – sera préférée ici par rapport à la XUP-V5 en raison de sa modernité (meilleur rendu vidéo, plus de crédibilité si l'approche cible du matériel récent, etc.).

Le choix de la plateforme, c'est-à-dire une architecture munie d'un potentiel de déploiement d'application avec un framework adapté, est maintenant figé. Nous allons grâce à cela passer au prochain chapitre à la mise en œuvre d'un exemple de système reconfigurable, dont la de conception va exploiter la méthodologie présentée dans cette étude.

8

Modélisation du cas d'étude

Sommaire

8.1 Introduction 157

8.2 Propriétés de l'exemple d'application 158

8.2.1 Contraintes 158

8.2.2 Besoins 159

8.3 Modélisation 160

8.3.1 Application 160

8.3.2 Architecture 167

8.3.3 Allocation 168

8.4 Partie manuelle 169

8.5 Conclusion 169

8.1 Introduction

Ce chapitre a pour objet de présenter un cas d'étude capable de mettre en œuvre les méthodes de contrôle exposées jusqu'ici, puis d'en donner la modélisation en RecoMARTE. Ce chapitre est particulièrement illustré dans la publication [54]. Il s'agit tout d'abord de rassembler les besoins et contraintes en terme de démonstration, tant au niveau *application*, qu'au niveau *architecture* et *allocation*, puis de décrire génériquement un exemple de système correspondant. Une fois les caractéristiques de ce système spécifiées, il s'agira ensuite de construire un modèle RecoMARTE approprié en procédant étape par étape et en montrant l'intégration du contrôle.

L'exemple proposé dans ce chapitre à notamment été publié dans [55], afin d'illustrer la modélisation par contrainte du contrôle de reconfiguration.

8.2 Propriétés de l'exemple d'application

8.2.1 Contraintes

L'exercice consiste ici à déterminer un exemple à la fois simple et complet pour la démonstration de la méthodologie. Une contrainte majeure à prendre en compte en amont de la conception est en effet la complexité des transformations en terme combinatoire. Afin de disposer d'un exemple aisé à suivre tout au long de sa conception jusqu'à son exécution, il est important de se limiter à la fois en terme de tâches, d'implémentations et de modes de fonctionnement. L'exemple doit donc être suffisament complet pour tout démontrer, mais suffisament concis pour être exploré en détails à toute phase de sa conception/transformation.

L'exemple montré dans ce chapitre est une proposition de démonstrateur à incrémenter par les autres membres du projet FAMOUS. Un démonstrateur plus conséquent sera proposé à terme dans le projet afin de présenter les autres aspects de conception abordés dans FAMOUS, mais ne servirait dans le cadre de cette étude seule qu'à démontrer des propriétés telles que la scalabilité de la solution, ou la performance d'exécution du contrôle.

Concernant la scalabilité, nous savons déjà que la solution proposée embarque des mécanismes de traitement à complexité exponentielle. La présente étude n'est pas une contribution dans le domaine de l'optimisation de ces traitements, c'est donc sans surprise que des modèles dont l'espace d'états est grand (au delà d'une centaine de milliers d'états dans le cas d'un traitement par une station de travail puissante) ne pourront être pris en charge par la solution. Ceci est un problème spécifique à la synthèse de contrôleur, or la synthèse du contrôle ne peut être traitée de manière maximalement permissive que par ce type d'algorithme (toute autre approche est potentiellement sous-optimale car non certifiée par synthèse). Une approche manuelle par exemple, même vérifiée, n'est de toute façon pas une alternative dans cette classe de problème.

Concernant la performance d'exécution, principalement celle des contrôleurs générés, nous sommes ici tributaires de la qualité (certes formelle) du code produit par le compilateur BZR. Là encore, cette étude n'est pas une contribution dans l'optimisation du compilateur. Cependant, nous savons que l'exécution d'un pas de contrôle repose sur le parcours d'une structure arborescente (de forme if-then-else) ; chaque nœud de l'arbre (évaluation d'une condition) étant prédéfini statiquement, les opérations élémentaires d'un parcours sont peu complexes en terme de temps d'exécution (il s'agit de vérifier si un booléen donné est vrai ou faux) ; et sachant que la traversée d'un tel arbre est sous-linéaire (de complexité $O(\log(n))$, où n est le nombre d'états), la complexité du contrôle risque assez peu d'être un problème comparée aux autres algorithmes avec lesquels un contrôleur collabore au sein du système, en particulier la décision, la membrane, etc. Ces derniers al-

gorithmes sont en effets plus susceptibles d'appartenir à une classe de complexité élevée, et donc d'être prioritaires à améliorer en cas de besoin d'optimisation.

L'optimisation en général n'est pas un problème traité dans cette étude qui se focalise essentiellement sur la mise en place d'un flot de conception embarquant une méthode formelle. Seules la simplicité du flot (simplification de la conception) et sa sécurité sont mises en avant. C'est pourquoi il est ici justifié de restreindre la taille du démonstrateur afin de montrer l'essentiel ; l'explosion combinatoire (dans la scalabilité) et la complexité d'exécution seront des problèmes rencontrés naturellement avec l'augmentation de l'espace d'états, jusqu'à un certain point (soutenable jusqu'à une centaine de milliers d'états) où il sera plus judicieux d'abandonner l'attrait du contrôle maximalement permissif pour se tourner vers d'autres solutions (mais sous-optimales).

8.2.2 Besoins

Puisqu'il est nécessaire de représenter des systèmes disposant de contraintes de contrôle impactant la composition synchrone d'éléments dynamiques (afin d'exploiter nos transformations à leur plein potentiel), le système de démonstration devra disposer :

- d'au moins deux tâches, disposant chacune de plusieurs implémentations (au moins deux), afin que le contrôle puisse s'exprimer sur l'instanciation dynamique des combinaisons d'implémentations (configurations effectives du système) ;
- d'au moins deux automates composés en parallèle, chacun pilotant respectivement le comportement d'une tâche donnée.

L'exemple devra également pouvoir illustrer des contraintes implicites de combinaisons d'implémentations (ne reposant donc pas sur leur désignation explicite), notamment grâce l'emploi de *poids*, c'est-à-dire de valeurs de propriétés non fonctionnelles ; il faudra pour cela :

- définir au moins deux types de NfpMeasures, avec leurs opérations et propriétés, afin de permettre un contrôle/décision sur plusieurs critères ;
- valuer ses NfpMeasures sur des Configurations MARTE représentant des mappages d'une implémentation vers une zone reconfigurable ;
- spécifier un objectif de contrôle exploitant les NfpMeasures valuées.

Enfin, afin de montrer un potentiel de décision, il sera nécessaire d'introduire une forme d'indéterminisme dans la spécification de contrôle, afin qu'un contrôleur généré puisse proposer plusieurs implémentations autorisées au prochain pas. À ce titre, il sera proposé :

- de spécifier au moins deux transitions sortantes d'un état vers deux autres états (un des automates devra donc avoir au moins trois états) ;

– d'exploiter le mécanisme de valuation des variables contrôlables afin que le contrôleur, lors d'une valuation, active plusieurs transitions à la fois et permette ainsi d'effectuer un choix.

8.3 Modélisation

Cette section fournit un modèle en RecoMARTE, répondant aux besoins et contraintes précédamment définis. Il s'agit d'un extrait d'un modèle plus complet (ce dernier disposeront de l'ensemble des spécifications des aspects du système), focalisé sur la notion de contrôle.

8.3.1 Application

Pour rappel, l'objectif du modèle d'application dans MARTE est de définir les fonctionnalités, c'est-à-dire les tâches d'un système donné ainsi que leurs communications.

Nous avons expliqué que la partie "communication" est abstraite dans notre méthodologie, qui est un module précis du flot FAMOUS. Cette partie étant à terme traitée par la membrane, nous l'implémenterons manuellement dans cet exemple pour le moment ; c'est pourquoi nous choisissons de rester sur un modèle de communication simple et régulier, à base de réseau de Kahn.

La spécification du modèle d'application se déroule en deux temps : l'élaboration d'un graphe de tâches dédiées au traitement de données, et la définition du contrôle (également via des tâches mais identifiées comme dédiées au contrôle).

Graphe de tâches

Nous proposons ici une application de traitement d'un flux vidéo, comprenant deux tâches reconfigurables, exécutées en séquence de telle manière que la première communique le résultat de son traitement à la deuxième.

Nous allons donc implémenter deux tâches, *Resolution* et *Filter*, dont les rôles respectifs sont de modifier la taille de l'image du flux vidéo et d'adapter ses couleurs.

La figure 8.1 donne la représentation en (Reco)MARTE de ces deux tâches.

Définition du comportement

Il s'agit maintenant de donner la définition du comportement de reconfiguration pour l'application. Nous élaborons d'abord les Configurations MARTE, définissant les implémentations possibles des tâches, cf. figure 8.2. Les tâches *Resolution* et *Filter* ont ainsi

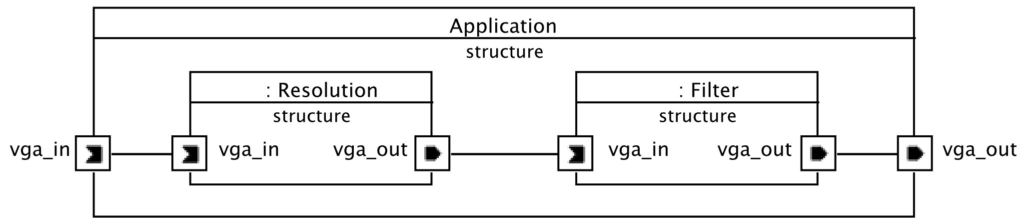


FIGURE 8.1 – Graphe de tâches représentant l'exécution en séquence de *Resolution* et *Filter*

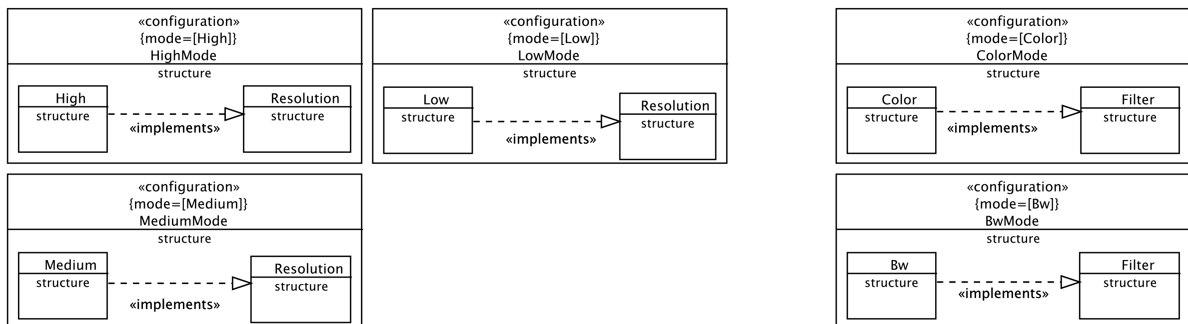


FIGURE 8.2 – Représentation des liaisons tâches/implémentations via des Configurations MARTE

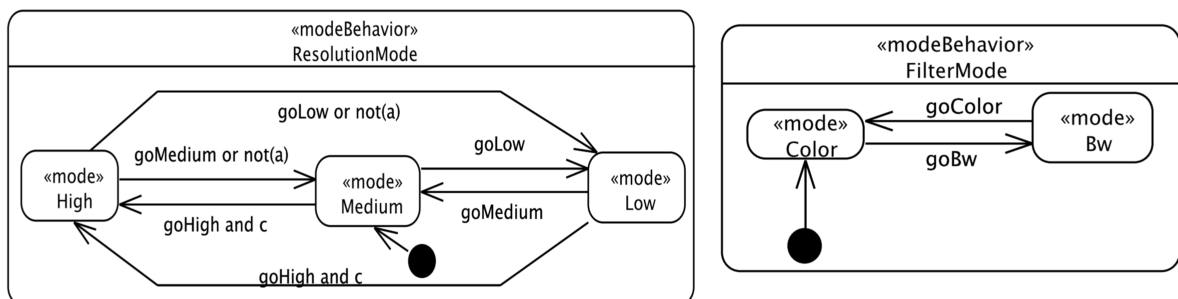


FIGURE 8.3 – Définition du comportement via des ModeBehaviors

respectivement trois implémentations (High, Medium, Low) et deux implémentations (Color, Bw) possibles.

Chaque Configuration ainsi définie est rattachée à un mode (donné par la propriété *mode* du stéréotype Configuration). Ces modes appartiennent à des ModeBehaviors, que nous proposons en figure 8.3 afin de représenter les conditions permettant de passer d'une Configuration MARTE à une autre, c'est-à-dire au final ce qui permet de passer d'une implémentation d'une tâche donnée à une autre dans le cadre d'une reconfiguration dy-

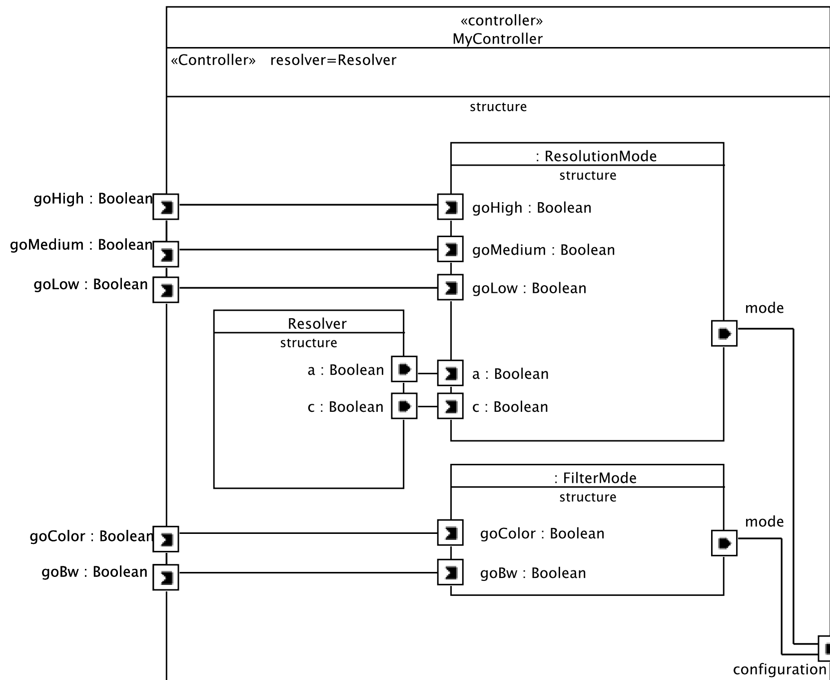


FIGURE 8.4 – Synchronisation des ModeBehaviors via un RtUnit dédié au contrôle

namique. Deux ModeBehaviors sont donc définis, *ResolutionFSM* et *FilterFSM*, et dédiés implicitement (et respectivement) aux comportements des tâches *Resolution* et *Filter* (via des liens indirectement établis entre les modes, les Configurations et les définitions d'implémentations). Sept événements booléens sont déclarés, sur occurrence desquels les ModeBehaviors peuvent réagir :

– *goColor* et *goBw* pour *FilterFSM*
– *goHigh*, *goMedium*, *goLow*, *a* et *c* pour *ResolutionFSM*

Ces ModeBehaviors sont pour l'instant spécifiés de manière indépendante. Mais afin d'illustrer la notion de contrainte sur la combinaison synchrone d'états actifs, nous allons les synchroniser en employant l'élément RecoMARTE *Controller*. Dont définissons à cet effet un RtUnit, c'est à dire une nouvelle tâche, que nous nommons "MyController", au sein duquel nous implantons les deux ModeBeahaviors (en vue composant) en synchronisant leurs entrées sur l'interface du RtUnit, cf. figure 8.4. Autrement dit dans cette sémantique, le RtUnit n'est exécutable que sur occurrence de l'ensemble des événements à un instant donné ; ainsi *ResolutionFSM* et *FilterFSM* peuvent réagir en même temps. Pour les événements *a* et *c*, nous n'allons pas les connecter à l'interface de la tâche MyController, car nous allons considérer que leur valuation n'est pas donnée par l'environnement, mais doit être calculée au sein de cette tâche. Nous exploitons alors notre extension RecoMARTE en posant le stéréotype *Controller* sur MyController, ce qui per-

met d'ajouter un nouveau composant à ce RtUnit, nommé *Resolver*, que nous identifions – via la propriété adéquate du stéréotype *Controller* – en tant que résolveur des valuations de a et c.

Le positionnement des variables contrôlables a et c n'est pas anodin. Il s'agit à terme d'illustrer les trois aspects essentiels de la notion de contrôle dans notre approche :

- la possibilité d'interdire une transition, par valuation de sa garde à *false* : si, depuis le mode Medium, c est valuée à *false* par le résolveur, alors quelle que soit la valeur de goHigh il sera impossible d'atteindre le mode High ;
- la possibilité de forcer une transition, par valuation de sa garde à *true* : si, depuis le mode High, a est valuée à *false* par le résolveur, il sera impossible de rester dans High quelles que soient les valeurs de goMedium et goBw ;
- ce qui amène au dernier aspect : l'exploitation du non-déterminisme par valuation à *true* de plusieurs gardes sortantes d'un état courant, ce qui implique la proposition d'un espace de configurations accessibles dans lequel il faudra choisir une évolution ; c'est le cas notamment depuis High, si a est valuée à *false*, ou si goMedium et goBw sont valués à *true* au même instant.

La valuation par défaut à *true* des variables contrôlables libres est ici clairement exploitée. Si a et c sont libres, alors leur valuation par défaut à *true* est comme sans effet sur la spécification de contrôle : aucune transition n'est forcée ou interdite, quelles que soient les valuations des autres variables et ce depuis n'importe quel état courant. En cas de besoin de contrôle (pour respecter une contrainte temporelle), et seulement en cas de besoin puisque le contrôleur est maximalement permissif, on comprend que la valuation imposée à a ou à c est nécessairement *false*, afin de provoquer le forçage, ou respectivement l'interdiction, d'une transition.

Cette méthodologie de spécification ainsi exemplifiée est généralisable. Soit E l'expression booléenne d'une garde, donnée sous forme normale conjonctive et composée des littéraux e_1, e_2, etc. tel que $E = e_1 \vee e_2 \vee \ldots$:

- soit c une variable contrôlable, il est possible de forcer un littéral e_n à être *false* – quelle que soit sa valuation d'origine – annulant ainsi sa prise en compte pour l'autorisation de la transition dont la garde est E en remplaçant e_n par $(e_n \wedge c)$; en effet, si c est libre donc *true*, alors c est sans effet car $(e_n \wedge c) = e_n$, mais si c est forcée à *false*, alors $(e_n \wedge c) = false$;
- soit a une variable contrôlable, il est possible de forcer un littéral e_n à être *true* – quelle que soit sa valuation d'origine – annulant ainsi sa prise en compte pour l'interdiction de la transition dont la garde est E en remplaçant e_n par $(e_n \vee \neg a)$; en effet, si a est libre donc *true*, alors a est sans effet car $(e_n \vee \neg a) = e_n$, mais si a est forcée à *false*, alors $(e_n \vee \neg a) = true$;

– il est également possible de combiner les effets de forçage et d'interdiction d'un littéral, en gardant le principe d'un contrôle sans effet lorsque toute variable contrôlable est libre : si a et c sont deux variables contrôlables alors en remplaçant e_n par $((e_n \wedge c) \vee \neg a)$ il est possible de forcer – respectivement interdire – e_n lorsque $a = false$ – respectivement lorsque $c = false$ – et si a et c sont libres, nous avons bien $((e_n \wedge c) \vee \neg a) = e_n$

À noter que notre manière d'exploiter le non-déterminisme par valuation de variables non-libres peut également être généralisée. Le principe de cette méthode est d'inciter le contrôleur à fournir un certain nombre de configurations alternatives dans le cas où le système est en échec par défaut (c'est-à-dire en prenant en compte une valuation par défaut pour toute variable contrôlable). Soit A l'état courant, et soient B_1, B_2, ... les états possédant une transition entrante depuis A. Lorsque A est interdit au prochain pas, et aucune transition sortante n'est activée par défaut (en laissant les variables contrôlables libres), alors nous pouvons provoquer une proposition de transitions forcées alternatives, grâce à un forçage approprié de variables contrôlables de la manière suivante : soient E_{k_1}, E_{k_2}, ... un ensemble de transitions sortantes vers B_{k_1}, B_{k_2} depuis A que nous voulons forcer, et a une variable contrôlable, il suffit de représenter chaque E_k sous la forme $E_k \vee \neg a$. Ainsi, si aucune transition n'est activée par défaut et si A est interdit au prochain pas, alors le contrôleur (s'il a pu être synthétisé) doit forcer a à false, ce qui provoque l'activation de toutes les transitions E_k et évite ainsi au système d'être en échec en permettant à un module de décision de choisir une alternative.

En suivant l'approche donnée précédemment, il est également possible de raffiner les propositions de configurations alternatives en forçant non plus des transitions complètes, mais seulement certains des littéraux de leurs gardes exprimées en forme normale conjonctive.

Définition des poids

Afin de permettre la définition de contraintes sur poids de configurations, nous allons ici exploiter notre méthodologie afin de 1) définir des types de poids et 2) attribuer des valeurs de poids aux configurations.

Pour rappel, nous avons augmenté MARTE afin d'autoriser la définition de propriétés non-fonctionnelles particulières, liées à des opérations permettant de les combiner et comparer : c'est le concept de NfpMeasure. Nous allons ajouter deux types de NfpMeasures à cet exemple : la notion d'énergie consommée, *power*, et la qualité de service, *qos*.

La figure 8.5 présente la spécification de ces deux types de NfpMeasures, que nous ajoutons à notre modèle d'application. Les valeurs que nous donnerons en terme de consommation et de qualité de service seront entières, c'est pourquoi les deux types héritent de

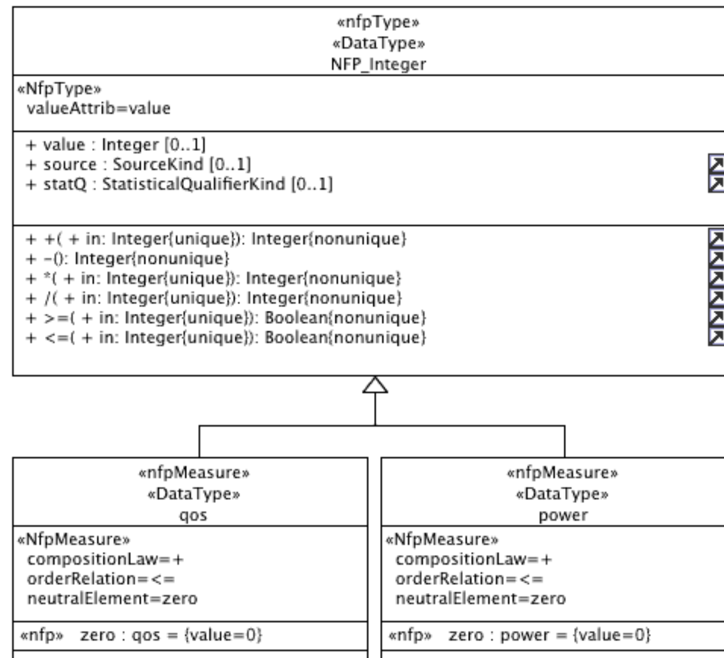


FIGURE 8.5 – Spécification des types de NfpMeasures *power* et *qos*

NFP_Integer, présent dans la librairie MARTE, afin de bénéficier de la définition des opérations sur entiers donnée pour NFP_Integer. La combinaison de valeurs *power* ou de valeurs *qos* s'effectuera par addition, c'est pourquoi l'addition héritée depuis NFP_Integer est associée à la propriété *compositionLaw* du stéréotype NfpMeasure pour ces deux Nfp-Measures. La comparaison permettant de classer deux valeurs de *power* ou de *qos* s'effectuera par l'opération ≤, héritée depuis NFP_Integer et associée la propriété *orderRelation* du stéréotype NfpMeasure. Enfin, lorsqu'aucune valeur ne sera fournie pour une Configuration MARTE à combiner, nous emploierons un élément dont la valeur sera neutre pour l'opération de combinaison. Cet élément désigne l'attribut *zero* pour chaque NfpMeasure, défini lui-même comme une instance valuée à 0 du type de la NfpMeasure qui le contient, et associé à la propriété *neutralElement* du stéréotype.

Il s'agit à présent d'attribuer des valeurs aux Configurations MARTE pour ces Nfp-Measure, ce qui est donné en figure 8.6 montrant les Configurations que nous avons définies précédemment en vue *classe* (et non *composite*). Le principe est d'ajouter des NFP en tant qu'attributs de ces configurations, de les typer selon le type de NfpMeasure choisie, et de fournir une valeur à l'attribut *value* (hérité de la définition de NFP dans MARTE pour NFP_Integer et donc hérité pour nos NfpMeasures) en respectant la syntaxe VSL. Bien entendu, pour des besoins de démonstration, les valeurs arbitraires que nous donnons ici serviront à introduire des propriétés intéressantes au niveau du contrôle. Sur un

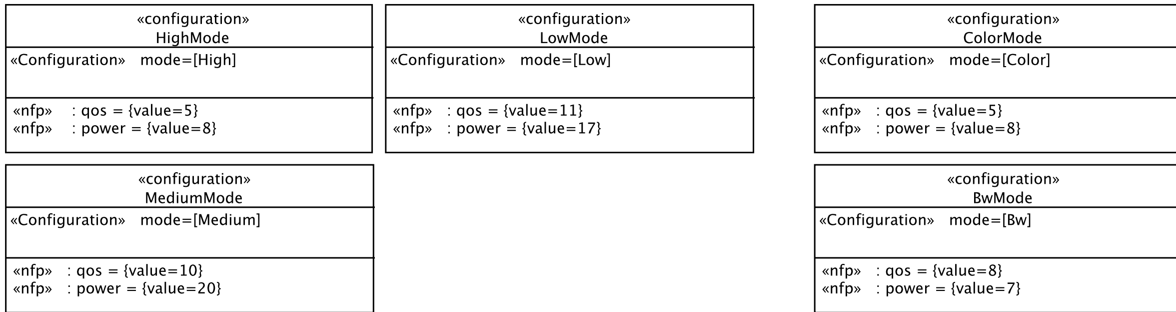


FIGURE 8.6 – Attribution de valeurs de NfpMeasures aux Configurations MARTE

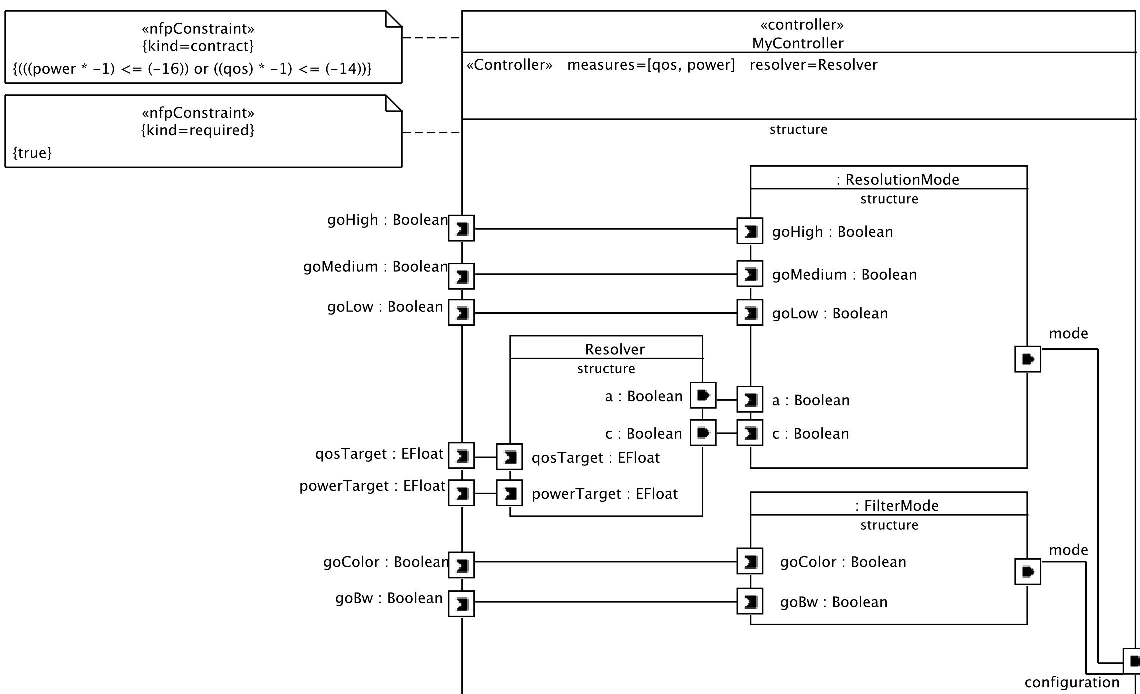


FIGURE 8.7 – Prise en compte des NfpMeasures pour ce contrôleur (déclaration de types, contrainte sur poids, décision)

exemple plus réel, ces valeurs seraient par exemple issues de l'évaluation des configurations spécifiées (par expertise, par profiling, etc.).

Contraintes temporelles du système

Nous allons maintenant pouvoir compléter le RtUnit dédié au contrôle que nous avons commencé en figure 8.4, en prenant en compte les poids (NfpMeasures) attribués aux Configurations MARTE dans le but de définir des contraintes temporelles.

La figure 8.7 montre le contrôleur ainsi complété. Tout d'abord les types de NfpMeasures à prendre en compte sont déclarés au moyen de la propriété *measure* du stéréotype *Controller*, on restreint ainsi les futures transformations de ce contrôleur à la seule prise en compte des NfPMeasures *prower* et *qos*. Afin de montrer que le mécanisme de décision après contrôle peut disposer d'une interface qui lui est propre, nous ajoutons également deux événements d'entrées au résolveur, qosTarget et powerTarget, de valeurs réelles et synchronisés sur les autres événements de ce contrôleur. Ainsi à chaque appel d'un pas de contrôle, des consignes de consommation et de qualité de service seront communiquées depuis l'environnement au système interne de décision de reconfiguration. Enfin, deux contraintes temporelles sont spécifiées et connectées au contrôleur :

- une NfpConstraint de type *required* d'après sa propriété, d'expression booléenne ($true$), spécifiant ainsi le fait que toute valuation des entrées est correcte pour tout exécution de ce contrôleur (l'expression est en effet toujours vraie quelles que soient les valeurs d'entrées) ;
- une NfpConstraint de type *contract* d'après sa propriété, d'expression booléenne $(((power * -1) \leq (-16)) \vee (qos * -1) \leq (-14))$, spécifiant le fait que le contrôleur doit s'engager à faire en sorte que la consommation d'énergie soit strictement supérieure à 16 unités ou bien que la qualité de service soit strictement supérieure à 14 unités.

8.3.2 Architecture

La sémantique restant encore à déterminer quant à la modélisation d'une architecture reconfigurable dans RecoMARTE, nous nous limiterons ici à supposer qu'il existera deux régions matérielles reconfigurables dans le modèle d'architecture, que nous nommerons *BBresolution* et *BBfilter* – assimilables à des *blackbox* dans la terminologie Xilinx – dont le but sera de supporter les Processing Elements (PE), c'est-à-dire les implémentations des tâches *Resolution* et *Filter* qui viennent d'être définies.

Concernant la génération de l'architecture, puisqu'il n'y a pas encore de modèle, c'est ici que nous faisons appel au framework pour la définition générique d'architectures de traitement vidéo que nous avons défini au chapitre précédent. La figure 8.8 montre l'architecture effective de cet exemple.

À noter que nous prenons ici une liberté particulière par rapport à la spécification du système (afin de rester synthétique) : la taille en entrée de l'image peut être différente de la taille en sortie à cause de la tâche *Resolution* ; nous ajoutons donc deux tâches *South* et *East* dont le rôle est simplement de créer des pixels noirs sur les zones non traitées en sortie par la tâche composite *Application*.

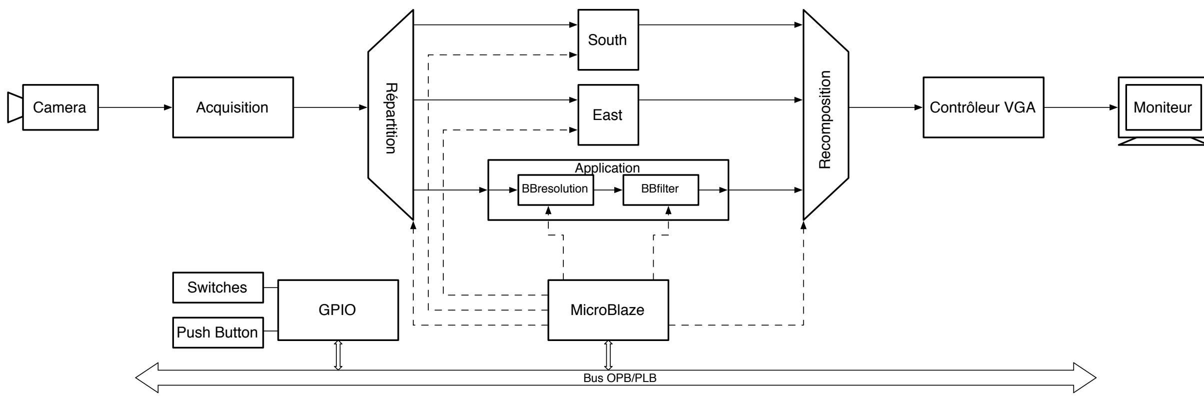


FIGURE 8.8 – Représentation du pipeline de traitement vidéo pour cet exemple de démonstration

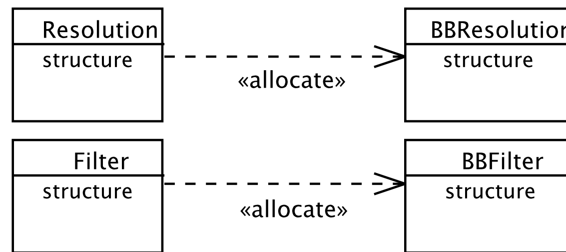


FIGURE 8.9 – Allocation des tâches du modèle d'application aux supposés éléments reconfigurables du modèle d'architecture

8.3.3 Allocation

De même pour l'allocation, puisque tous les éléments de l'architecture ne sont encore prêts à ce stade du projet, le modèle d'allocation ne peut être fourni de manière complète. Néanmoins, en partant de l'hypothèse de la présence de deux éléments matériels reconfigurables comme nous l'avons évoqué, nous pouvons déjà établir le modèle d'allocation en assimilant ces éléments à des composants abstraits.

La figure 8.9 montre ainsi l'allocation des tâches abstraites *Resolution* et *Filter* à leur blackbox respective *BBresolution* et *BBfilter*. Cela signifie ainsi – par extension depuis le modèle d'application – que les implémentations concrètes de *Resolution* et *Filter* sont instanciables sur leur blackbox allouée.

8.4 Partie manuelle

Comme nous l'avons dit, la génération de l'ensemble du système n'est pas traitée dans cette étude car seule la partie contrôle fait l'objet de la contribution. À terme donc dans le projet FAMOUS il sera possible de générer la partie architecture, via une extension appropriée de MARTE. Idéalement, l'ensemble de l'application devrait également pouvoir être généré, avec un modèle d'application RecoMARTE adéquat contenant une sémantique de traitement de données pour chaque tâche et implémentation, via par exemple le package Repetitive Structure Modeling (RSM) dans MARTE – permettant la modélisation complète d'applications de traitement intensif de données – le tout combiné à notre approche de modélisation du contrôle.

Ne disposant pour le moment ni d'un métamodèle pour la spécification d'architectures reconfigurables, ni de transformations d'une sémantique RSM vers un code exécutable, nous nous focaliserons simplement sur notre contribution – c'est-à-dire la génération du contrôle – et implémenterons l'outillage nécessaire à l'exécution complète grâce à la plateforme existante et à son framework exposés au chapitre précédent.

Pour la suite donc, nous allons partir du principe que nous avons codé manuellement chaque implémentation des tâches *Resolution* et *Filter*, et que nous disposons d'un programme s'exécutant sur le MicroBlaze (dédié à la reconfiguration) dont le principe est d'effectuer une boucle infinie au sein de laquelle des événements sont captés depuis l'environnement, puis un appel à la fonction – que nous appellerons *step* – exécutant un pas de contrôle est lancé avec ces événements, et enfin le résultat de cette fonction (un vecteur de booléens dont un seul est vrai) est traduit automatiquement en un vecteur d'identifiants de bitstreams à télécharger, lequel sert à appeler la fonction *loadbitstream* présente dans le framework afin de mettre en œuvre les reconfigurations appropriées. À noter que l'appel à *loadbitstream* reconfigure une zone *au bon moment*, c'est-à-dire lorsque les traitements effectués par le PE actif de cette zone sont terminés. C'est pourquoi dans ce cas simple d'exécution – où les tâches sont répétitives et ne sont pas supposées être suspendues – nous pouvons ici nous affranchir d'implémenter manuellement la *membrane* du projet FAMOUS, tout en gardant une exécution correcte.

8.5 Conclusion

Un extrait de modèle RecoMARTE, axé sur l'aspect contrôle, vient d'être spécifié pour un exemple particulier capable de démontrer la contribution de cette étude. Une fois la partie à réaliser manuellement implémentée sur la plateforme (bitstreams partiels et boucle globale d'exécution), il ne reste plus qu'à générer la partie contrôle de cet exemple, c'est-à-

dire le code de la fonction *step* appelé dans la boucle, qui réalise le traitement d'événements reçus, construit un ensemble de configurations accessibles, choisit l'une d'entre elles et produit un ordre de reconfiguration.

La génération et l'évaluation de la partie contrôle font l'objet du prochain chapitre.

9

Automatisation du flot

Sommaire

9.1 Introduction 172
9.2 Génération de la partie synchrone 173
9.2.1 Interface du nœud de contrôle MyController 173
9.2.2 Hypothèse 175
9.2.3 Objectif de contrôle 177
9.2.4 Définition de l'accessibilité des états 177
9.2.5 Définition de l'accessibilité des configurations 178
9.2.6 Automates réagissant d'après la décision 178
9.2.7 Calcul et attribution des combinaisons de poids 178
9.2.8 Compilation et synthèse 180
9.3 Génération du support logiciel pour l'implémentation de la décision 181
9.3.1 Génération de l'interface de décision 181
9.3.2 Accès aux métriques des configurations 183
9.4 Support pour la simulation 186
9.4.1 Communication vers le contrôleur 187
9.4.2 Visualisation de l'exécution du contrôle 187
9.5 Exécution du démonstrateur 189
9.5.1 Simulation 189
9.5.2 Chaîne de simulation 193
9.5.3 Intégration dans la plateforme 198
9.6 Conclusion 200

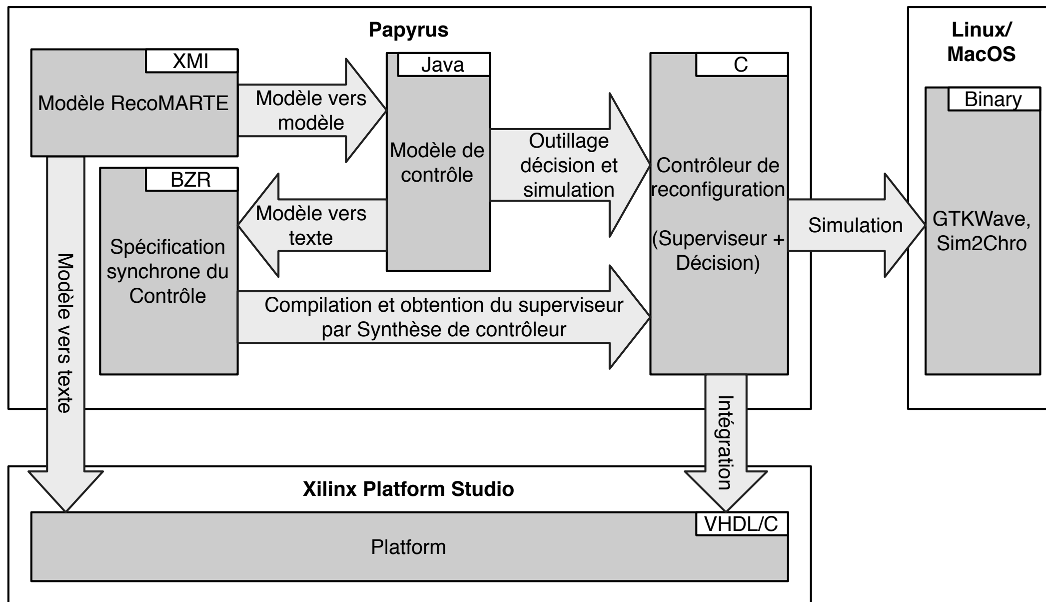


FIGURE 9.1 – Flot de conception, transformation, simulation et intégration pour la partie contrôle

9.1 Introduction

Ce chapitre final a pour objectif de présenter l'automatisation de la chaîne de transformations – jusqu'ici présentée sous un aspect théorique aux chapitres 5 et 6 – puis de procéder à la simulation du code de contrôle obtenu et à son intégration vers la plateforme reconfigurable.

La figure 9.1 montre ce flot dans sa globalité. Nous retrouvons en entrée un modèle RecoMARTE tel que celui que nous venons d'élaborer au chapitre précédent, au format XMI (standard XML pour la représentation de modèles), et élaboré avec le modeleur UML Papyrus (dans lequel nous avons également spécifié l'extension RecoMARTE). Vient ensuite une première transformation, de type *modèle vers modèle*, ciblant une représentation en modèle intermédiaire dont le métamodèle a été donné au chapitre 5. Puis une deuxième transformation traite ce modèle afin de produire le programme synchrone de contrôle, conformément aux spécifications données au chapitre 5 concernant la transformation d'un modèle intermédiaire vers BZR.

Le choix technologique étant libre dans le cadre de FAMOUS pour la réalisation d'un prototype de flot, nous avons recherché un outil permettant une implémentation à la fois simple et minimaliste, dans un souci de prototypage rapide capable de démontrer le flot dans des délais assez brefs. L'outil SDMetrics [127] a répondu à l'ensemble de ces critères. Il s'agit à la base d'un outil commercial dédié à l'évaluation de la qualité de modèles UML. Les fonctionnalités maîtresses de SDMetrics, l'*Open Core*, sont cependant

libres et nous intéressent particulièrement dans le sens où elles permettent la définition de métamodèles en vue d'opérer des requêtes (et donc des transformations) sur des instances (modèles) de ses métamodèles. Le moteur de requêtes de SDMetrics s'avère très pratique pour opérer des transformations particulières, notamment celles ayant besoin de parcourir plusieurs fois le modèle, et de manière non linéaire (élaboration de configurations à partir des compositions d'états, associations de poids à un état, etc.).

Une fois ce modèle intermédiaire établi par transformation depuis RecoMARTE, il devient possible de générer un code BZR approprié en le *visitant*[36] suivant les règles de transformations évoquées au chapitre 5, avec intégration de la partie décision. À ce stade, la représentation synchrone obtenue en BZR peut être compilée, faisant ainsi appel à la synthèse de contrôleur via SIGALI afin d'obtenir (en langage C) le superviseur maximalement permissif du système rendu déterministe via la méthode d'oracles. Mais le code C produit par cette étape nécessite d'être instrumenté afin d'être simulé ou intégré sur une plateforme. En effet, les routines de récupération d'événements en provenance de l'environnement ainsi que l'interface de décision ne sont pas prises en charge lors de la compilation d'un programme BZR ; une transformation complémentaire depuis le modèle intermédiaire vient donc compléter le code C du superviseur à cet effet.

D'autres transformations, abstraites par la flèche directe entre RecoMARTE et la plateforme dans la figure, relèvent de la responsabilité d'autres modules dans FAMOUS et ne sont pas présentées dans cette étude.

Dans la suite, nous allons nous focaliser sur les détails de génération de la spécification synchrone du contrôleur de reconfiguration, puis – après compilation BZR, synthèse, outillage, et compilation finale – sur ceux de sa simulation et de son intégration dans la plateforme reconfigurable.

Par ailleurs, la démonstration du flot de conception – illustrant par l'exemple l'ensemble de la méthodologie proposée ici – a fait l'objet de la publication [54].

9.2 Génération de la partie synchrone

Cette section décrit les différentes parties du programme BZR résultant des transformations opérées sur la spécification RecoMARTE donnée au chapitre précédent.

9.2.1 Interface du nœud de contrôle MyController

Le programme 13 montre la structure du fichier BZR généré, contenant un nœud `MyController` correspondant à la spécification synchrone du RtUnit correspondant d'après

36. cf. Design Pattern "Visiteur" [42]

```
1  node MyController (
2    (* Événements de l'environnement *)
3    goHigh,goMedium,goLow,goColor,goBw:bool;
4    (* Événements du système de décision *)
5    dMediumBw,dMediumColor,dLowBw,dLowColor,dHighBw,dHighColor:bool)
6  returns (
7    (* Production de l'ensemble des configurations accessibles *)
8    canMediumBw,canMediumColor,canLowBw,canLowColor,canHighBw,
9    canHighColor:bool;
10   (* Événements facultatifs (aide au suivi d'exécution) *)
11   enforceRule,assumeRule:bool;
12   power,qos:int)
13 contract
14   assume (assumeRule)
15   enforce (true -> pre(enforceRule))
16   with (a,c:bool)
17 var
18   (* Déclaration des variables internes *)
19   last medium:bool = false;
20   last low:bool = false;
21   last high:bool = false;
22   last bw:bool = false;
23   last color:bool = false;
24 let
25   (* Spécification de l'hypothèse, cf. programme 14 *)
26   (* Spécification de l'objectif de contrôle, cf. programme 15 *)
27   (* Définition de l'accessibilité des états, cf. programme 16 *)
28   (* Définition de l'accessibilité des configurations, cf. programme 17 *)
29   (* Automates réagissant sur événements de décision, cf. programme 18 *)
30   (* Calcul des combinaisons de poids, cf. programme 19 *)
31 tel
```

Algorithme 13: Structure du programme BZR généré d'après la spécification RecoMARTE du démonstrateur.

le modèle RecoMARTE. L'interface est ici exposée, présentant les entrées non contrôlables – composées des événements de l'environnement (préfixés par "go") et du système de décision (préfixés par "d") – ainsi que les sorties produites à chaque pas d'exécution – comprenant l'ensemble des booléens caractérisant l'accessibilité ou non d'une configuration correspondante (préfixés par "can") mais également l'état des objectifs et hypothèses de contrôle et la valeur des poids de la configuration courante (ces événements supplémentaires en sortie sont facultatifs mais prévus dans les transformations afin de faciliter le suivi d'exécution) –. Cette interface est produite en conformité avec la transformation spécifiée par l'équation 6.1 (p. 130).

La déclaration du *contrat* BZR – c'est-à-dire l'hypothèse et l'objectif de contrôle ainsi que les variables contrôlables – est également préparée ; nous retrouvons ainsi la déclaration des variables a et c, et nous verrons plus loin la valuation effective des variables `assumeRule` et `enforceRule` respectivement utilisées en tant qu'hypothèse (`assume`) et objectif de contrôle (`enforce`). À noter que la définition `enforce(true -> pre(enforceRule))` relève d'un détail technique propre à BZR traitant le cas particulier de l'initialisation du système, ce qui permet à la synthèse de contrôleur d'assurer `enforceRule` en partant du principe que l'objectif de contrôle est correct (*true*) au premier pas, mais vérifiera seulement au pas suivant que le pas précédent était effectivement correct (d'où l'emploi de l'opérateur `pre` à cet effet) ; ceci est obligatoire afin de contourner des problèmes de causalité survenant au premier pas, dus à la réception de la décision alors que le superviseur n'a pas encore émis d'espace de configurations accessibles.

Une troisième partie visible dans ce programme est la déclaration des variables internes, constituée des variables définissant l'activation courante d'un état (déclaration des variables b_q pour l'équation 6.3 p. 130). Nous retrouvons donc une variable de ce type par état déclaré. La syntaxe `last X:bool = false;` signifie que `X` est initialisée à *false* par défaut.

Tout autre composant de ce programme est exposé dans la suite par un sous-programme correspondant dont le numéro d'identifiant est mentionné en commentaire.

9.2.2 Hypothèse

La génération de l'hypothèse complète est donnée dans le programme 14, exposant la transformation donnée par les équations 6.5, 6.6 et 6.7 (p. 131). Elle est composée de l'hypothèse arbitraire donnée dans RecoMARTE par le concepteur et de deux règles telles que *au moins* et *au plus* une configuration est choisie par le système de décision parmi les configurations accessibles indiquées par le superviseur au pas précédent. On remarque que seule la configuration $\{Medium; Color\}$ est supposée être choisie au premier pas, ce qui correspond à la spécification RecoMARTE dans laquelle les états $Medium$ et $Color$ sont

```
1  (* Combinaison de l'hypothèse arbitraire (ici donnée à true) et des deux règles
   sur la décision *)
2  assumeRule = true & atLeastOne & atMostOne;
3  (* Au moins une configuration... *)
4  atLeastOne = ((false -> pre(canHighColor)) & dHighColor) &
5               ((true -> pre(canMediumColor)) & dMediumColor) &
6               ((false -> pre(canLowColor)) & dLowColor) &
7               ((false -> pre(canHighBw)) & dHighBw) &
8               ((false -> pre(canMediumBw)) & dMediumBw) &
9               ((false -> pre(canLowBw)) & dLowBw);
10 (* ... Et au plus une seule doit être sélectionnée par la décision *)
11 atMostOne = not(
12    (dHighColor & (dMediumColor or dLowColor or dHighBw or
13    dMediumBw or dLowBw)) or
14    (dMediumColor & (dLowColor or dHighBw or dMediumBw or
15    dLowBw)) or
16    (dLowColor & (dHighBw or dMediumBw or dLowBw)) or
17    (dHighBw & (dMediumBw or dLowBw)) or
18    (dMediumBw & (dLowBw)));
```

Algorithme 14: Spécification de l'hypothèse comportementale de l'environnement (et donc de la décision).

les états initiaux de leur ModeBehavior respectif, c'est-à-dire qu'il s'agit de la configuration initiale du système.

```
1 (* Combinaison de l'objectif arbitraire arbitraire et de l'objectif générique *)
2 enforceRule =
3     (* Partie générique... *)
4     (canBwHigh or canBwLow or canBwMedium or canColorHigh or
5     canColorLow or canColorMedium) &
6     (* ... Et partie arbitraire *)
7     (((power * -1) <= -16) or ((qos * -1) <= -14)));
```

Algorithme 15: Spécification de l'objectif de contrôle.

9.2.3 Objectif de contrôle

L'objectif de contrôle complet est donné dans le programme 15, issu de la transformation 6.4 (p. 130). Il est composé de l'objectif arbitraire donné par le concepteur dans MARTE et de l'objectif générique consistant à assurer qu'à chaque pas, au moins une configuration est accessible.

9.2.4 Définition de l'accessibilité des états

```
1 canHigh   = (medium & (goHigh & c)) or (low & (goHigh &
2             c)) or (high & not((goMedium or not(a)) or
3             (goLow or not(a))));
4 canMedium = (high & (goMedium or not(a))) or (low &
5             (goMedium)) or (medium & not((goHigh & c) or
6             (goLow)));
7 canLow    = (medium & (goLow)) or (high & (goLow)) or (low
8             & not((goMedium) or (goHigh & c)));
9 canColor  = (bw & (goColor)) or (color & not(goBw));
10 canBw     = (color & (goBw)) or (bw & not(goColor));
```

Algorithme 16: Définition de l'accessibilité des états.

L'accessibilité des états, donnée par une représentation équationnelle des automates est montrée par le programme 16, résultant de la transformation 6.3 (p. 130). Le principe est qu'un état est accessible si l'état courant dispose d'une transition potentielle vers cet

état. Pour représenter cela, nous nous servons des variables locales reflétant l'activation des états (initialisées dans la suite) que nous combinons avec les gardes des transitions pour former des équations d'accessibilité.

9.2.5 Définition de l'accessibilité des configurations

```
1 canMediumBw = canMedium & canBw;
2 canMediumColor = canMedium & canColor;
3 canLowBw = canLow & canBw;
4 canLowColor = canLow & canColor;
5 canHighBw = canHigh & canBw;
6 canHighColor = canHigh & canColor;
```

Algorithme 17: Définition de l'accessibilité des configurations.

La génération des équations d'accessibilités des configurations se sert ensuite des accessibilités d'états précédemment définies, cf. programme 17, conformément à la transformation 6.2 (p. 130). Il s'agit ici d'établir les combinaisons adéquates d'accès aux états correspondant à chaque configuration dont l'accessibilité est évaluée.

9.2.6 Automates réagissant d'après la décision

Les automates de la spécification d'origine sont ensuite modifiés pour réagir uniquement d'après les événements provenant du système de décision (préfixés par "d"), cf. programme 18. Le principe est de garder les états ainsi que les cibles et origines de toutes transition, mais de modifier leurs gardes afin de refléter la réaction sur décision, conformément à la transformation des transitions originales $\mathcal{T}_k$ en $\mathcal{T}'_k$ montrée par l'équation 6.8 (p. 132). À noter qu'à chaque définition d'état dans ces automates générés, la variable correspondante reflétant son activation est valuée à *true*, conformément à la valuation des variables b_q définie par cette même équation.

9.2.7 Calcul et attribution des combinaisons de poids

La dernière partie du programme BZR généré consiste en la spécification des poids sur configurations. Il s'agit dans un premier temps de les calculer, puis de les assigner aux variables de poids (ici : *power* et *qos*).

Sous réserve d'avoir implémenté les opérations sur poids (NfpMeasures) mentionnées dans le modèle RecoMARTE, conformément à l'approche décrite dans la sous-section5.2.2,

```
1  automaton
2    state Color
3      do
4      (bw,color) = (false,true);
5      unless dMediumBw or dLowBw or dHighBw then Bw
6
7    state Bw
8      do
9      (bw,color) = (true,false);
10     unless dMediumColor or dLowColor or dHighColor then Color
11
12 end;
13 automaton
14   state Medium
15     do
16     (medium,low,high) = (true,false,false);
17     unless dLowBw or dLowColor then Low |
18            dHighBw or dHighColor then High
19
20   state Low
21     do
22     (medium,low,high) = (false,true,false);
23     unless dMediumBw or dMediumColor then Medium|
24            dHighBw or dHighColor then High
25
26   state High
27     do
28     (medium,low,high) = (false,false,true);
29     unless dMediumBw or dMediumColor then Medium|
30            dLowBw or dLowColor then Low
31
32 end;
```

Algorithme 18: Automates d'origine, réagissant maintenant d'après les événements en provenance du système de décision.

```
1 (power,qos) =
2    if(dMediumBw) then (27,18) else (
3    if(dMediumColor) then (28,15) else (
4    if(dLowBw) then (24,19) else (
5    if(dLowColor) then (25,16) else (
6    if(dHighBw) then (15,13) else (
7    if(dHighColor) then (16,10) else (
8    (0,0)))))));
```

Algorithme 19: Attribution des poids aux configurations et définition de l'équation assignant le poids courant aux variables appropriées (*power* et *qos*).

il s'agit de les invoquer lors de l'établissement des combinaisons possibles d'états formant les configurations afin de calculer les poids statiques de toute configuration. L'assignation des poids dans BZR suit ici un modèle en *if-then-else* reposant sur l'évaluation des états activés. Lorsque tous les états d'une configuration donnée sont activés dans un pas d'exécution, les poids calculés pour celle-ci sont attribués aux variables (*power* et *qos*) définissant les poids courant du système. Cette assignation est montrée par le programme 19, reflétant la transformation donnée par l'équation 5.1 (p. 117). Pour rappel, l'opération de combinaison définie pour *power* et *qos* est l'addition sur les entiers. Pour chaque configuration du système, les valeurs attribuées aux Configurations MARTE (et donc indirectement aux états) sont alors additionnées suivant sa combinaison d'états. Par exemple, les valeurs respectives de *power* et *qos* sont de 20 et 10 unités pour $Medium$, et de 7 et 8 unités pour Bw, ce qui produit ainsi un poids final de 27 et 18 pour la configuration $\{Medium; Bw\}$ en appliquant les opérations de combinaison de *power* et *qos*.

À noter qu'une et une seule configuration est toujours active à tout pas d'exécution, sinon la synthèse de contrôleur échoue en amont pour cause de non respect des objectifs de contrôle, donc la valuation de *power* et *qos* par les valeurs neutres $(0, 0)$ n'aura jamais lieu et n'est ici présente que pour une raison de syntaxe.

9.2.8 Compilation et synthèse

La compilation de ce programme BZR – incluant une étape de synthèse de contrôleur afin de valuer les variables contrôlables a et c de manière appropriée d'après les contraintes spécifiées – sert à produire une fonction en C dont l'interface correspond à celle du nœud de la spécification BZR. Des fichiers Makefile sont également générés pendant les transformations afin de simplifier l'étape de compilation pour le concepteur, il s'agit en effet par la suite d'intégrer le résultat de compilation produit par BZR à un mécanisme de dé-

cision ainsi qu'à du code d'instrumentation pour simulation, puis de compiler le tout en disposant d'une structure déjà clairement définie.

Pour l'instant, en lançant la compilation du code BZR de cet exemple, celle-ci parvient à son terme ce qui signifie que 1) il existe bien un superviseur concret pour notre spécification, et 2) son code vient d'être produit par synthèse, de manière à la fois correcte et maximalement permissive ; il est accompagné de la routine à base d'*oracles* pour le rendre déterministe. À noter que lorsque la compilation échoue à ce niveau, il est alors certain qu'aucun superviseur n'existe dans ce cas, et il est alors nécessaire de revoir soit le système, soit ses contraintes.

Le code résultant de la compilation BZR nécessite maintenant d'être outillé de manière à pouvoir récupérer des événements, intégrer un système de décision, et permettre la simulation purement logicielle du contrôleur en communiquant via des outils appropriés.

9.3 Génération du support logiciel pour l'implémentation de la décision

L'objectif étant de transformer cette fonction générée de manière à ce qu'elle ne prenne en entrée que les événements de l'environnement et ne renvoie plus qu'une seule configuration à chaque pas d'exécution (afin d'être conforme à la spécification RecoMARTE, dans laquelle un RtUnit de contrôle capte des événement non contrôlables et émet une seule configuration à chaque exécution), il est nécessaire de l'encapsuler dans une nouvelle fonction, possédant l'interface suivante :

- événements non contrôlables en entrée ;
- identifiant de configuration en sortie.

Afin que cette fonction puisse exécuter le superviseur, elle doit également encapsuler un mécanisme de décision intervenant après chaque pas de contrôle afin de d'enregistrer son choix à la fois en tant qu'ordre de reconfiguration à émettre et en tant qu'entrée du superviseur pour le prochain pas. La fonction doit donc capter les événements non contrôlables, et les fournir au superviseur accompagnés de la décision au pas précédent (en générant le cas de la décision initiale, coïncidant avec la configuration de départ prévue pour le système d'après la spécification RecoMARTE, ici $[Medium; Color]$).

9.3.1 Génération de l'interface de décision

L'implémentation même de la décision relève de la responsabilité du concepteur. Toutefois, la génération de son interface peut déjà être établie car elle est supposée se conformer à celle du superviseur. Soient `res` et `choice` les noms donnés à deux variables dont

le type est une structure (nommée `MyController_res`) composée d'un ensemble de booléens (un booléen par configuration). Un pas de contrôle global comprend dans un premier temps un appel à un pas du superviseur qui vient valuer `res`, contenant ainsi l'espace de configurations accessibles au prochain pas.

Vient ensuite l'appel au mécanisme de décision, qui doit choisir un seul booléen valué à *true* dans `MyController_res`. Ce choix est placé dans `choice` qui sera traduit ensuite en identifiant de configuration choisie.

```
1 int Ctrlignore_decide(MyController_res* res, MyController_res*
  choice, int* lastChoice, float* powerTarget, float* qosTarget) {
2
3   /* Recherche dynamique de la frontière de Pareto de l'ensemble
    des configurations accessibles */
4   MyController_buildPareto(res, choice, lastChoice);
5
6   /* Mécanisme de décision généré par défaut, opérant un choix
    aléatoire */
7   *lastChoice = MyController_okConfigsIDs[rand() %
    MyController_nbOkConfigs];
8
9   /* Mise à jour de la décision, valuation de choice pour le
    prochain pas */
10  MyController_setConfig(choice, lastChoice);
11
12  /* Retour de l'identifiant de la configuration choisie, à
    émettre vers le système pour reconfiguration */
13  return *lastChoice;
14 }
```

Algorithme 20: Génération de l'interface de décision, avec mécanisme de décision par défaut à adapter.

L'interface de décision générée pour ce démonstrateur est donnée dans le programme 20. L'attribut `lastChoice` sert ici simplement à effacer l'ancienne valeur choisie et enregistrée dans `choice`, afin de remettre `choice` à zéro à chaque pas. On remarque également la présence des deux entrées dédiées à la décision d'après la spécification RecoMARTE : `powerTarget` et `qosTarget`.

Cette interface générée, disposant d'une implémentation simple, est compilable et im-

médiatement exécutable. Sans rentrer dans les détails techniques d'implémentation, son principe de fonctionnement est le suivant :

- la frontière de Pareto des configurations accessibles est établie dynamiquement d'après `res` dans un tableau d'identifiants de configurations correspondant, ici nommé `MyController_okConfigsIDs` ;
- un mécanisme par défaut de choix de configuration est effectué, en sélectionnant aléatoirement l'indice d'une case du tableau de la frontière de Pareto ;
- la variable `choice` est mise à jour, elle correspond à l'ensemble des booléens de décision à donner en entrée au superviseur à chaque pas d'exécution ;
- l'indice de la configuration choisie est retourné, il pourra ensuite être émis vers le système afin qu'il se reconfigure en conséquence.

Diverses fonctions et structures sont également générées afin de servir de support au concepteur dans le cadre de l'implémentation de la décision. C'est notamment le cas :

- de la fonction de Pareto comme nous venons de le voir, qui a été spécifiée génériquement dans l'algorithme 20 ;
- de la fonction de contrôle proportionnel et intégral (PI) que nous avons mentionnée précédemment à titre d'exemple de mécanisme de décision (seuls les coefficients de l'algorithme restent à être optimisés par le concepteur) ;
- du vote de Borda permettant d'effectuer une optimisation multi-critères
- de structures permettant l'accès aux métriques des configurations ; celles-ci servent notamment à la fonction de Pareto (treillis) et au vote de Borda (tableau ordonné de configurations pour chaque poids).

9.3.2 Accès aux métriques des configurations

Si la génération des fonctions génériques de Pareto, contrôle PI, et vote de Borda suit sans surprise les indications de fonctionnement données au chapitre 5, les détails de génération des structures d'accès aux métriques méritent d'être observés.

Tableau triés de configurations

Pour chaque type de poids, on génère un tableau à une dimension dont l'indice de chaque case correspond à l'identifiant d'une configuration, et la valeur de chaque case désigne la valeur de poids de la configuration associée. Soient `MyController_nbConfigs` le nombre de configurations gérées par le présent contrôleur et `MyController_power` et `MyController_qos` deux tableaux de ce type pour les poids respectifs *power* et *qos*, leurs déclarations statiques pour le démonstrateur sont les suivantes :

```
int MyController_power[MyController_nbConfigs] = {27,15,24,28,16,25};
```

```
int MyController_qos[MyController_nbConfigs] = {18,8,19,15,5,16};
```

Les identifiants des configurations, correspondant à leur place dans le tableau myController_res, sont les suivants :

```
[Bw;Medium] : 0
[Bw;High] : 1
[Bw;Low] : 2
[Color;Medium] : 3
[Color;High] : 4
[Color;Low] : 5
```

Afin d'ordonner le tout, un tableau supplémentaire est généré, dont l'indice de chaque case désigne la place recherchée pour une configuration d'après un poids (la place d'indice le plus élevé est la meilleure, le tri étant opéré d'après l'opération d'ordre spécifiée par le concepteur pour chaque poids). La valeur de chaque case est un tableau dont les indices sont les identifiants de poids (*power* :0 et *qos* :1) et les valeurs sont les indices des cases dans les tableaux de poids respectifs (`MyController_power` et `MyController_qos`), coïncident avec des identifiants de configurations. Soit `MyController_nbWeights` le nombre de poids gérés par le présent contrôleur, ce tableau se définit donc ainsi :

```
int MyController_weights[MyController_nbConfigs][MyController_nbWeights]
= {{1,4},{4,1},{2,3},{5,5},{0,0},{3,2}};
```

D'après le tableau, les configurations n°1 ({Bw ;High}) et 4 ({Color ;High}) sont donc celles ayant le poids le moins intéressant respectivement pour *power* (15 unités) et *qos* (10 unités), etc. Ces tableaux peuvent ensuite être exploités par le concepteur dans le cadre de l'implémentation de la décision. Il sont par défaut utilisés dans la fonction de vote de Borda, dans laquelle l'établissement des votes (où les candidats sont les configuration et les électeurs sont les poids) s'effectue en parcourant dans l'ordre le tableau `MyController_weights` à la recherche de la configuration la plus proche d'une consigne pour chaque poids.

Treillis de configurations

Suivant l'algorithme donné au chapitre 5 concernant la mise à plat du treillis de configurations (contenu dans le modèle intermédiaire), nous obtenons le tableau à une dimension qui suit pour ce démonstrateur :

```
int MyController_lattice[] = {
    3,
    0,1,5,
    2,2,1,4,
    3,2,1,4,

    3,
    5,2,1,4,
    1,0,
    4,0,
    0};
```

Pour rappel du principe, la première case désigne le nombre de configurations non optimisables par d'autres configurations (il s'agirait donc de la frontière de Pareto statique si toute configuration était permise), il y en a ici 3, d'identifiant 0, 2 et 3. Pour la première configuration d'identifiant 0, celle-ci optimise immédiatement une seule configuration, la n°5, d'où la valuation des trois cases suivantes : 0 (identifiant), 1 (nombre de configurations inférieures), 5 (identifiant de la seule configuration inférieure). Ce principe de mise à plat continue ainsi jusqu'à ce que toutes les configurations du treillis soient parcourues, terminant ainsi le tableau par 0, ce qui indiquera lors de son parcours que plus aucune configuration n'est à traiter.

Comme pour les tableaux triés de configurations, ce treillis peut servir au concepteur dans l'élaboration de la décision. Il est notamment employé par la fonction établissant la frontière de Pareto dynamique. À titre d'exemple, si pour un pas donné les configurations n°1, 3 et 5 sont accessibles, alors d'après la représentation en une dimension du treillis, la fonction devra retourner une frontière composée seulement des configurations n°3 et 5, car la configuration n°1 est optimisée par au moins une autre configuration accessible. Ceci est représenté graphiquement par la figure 9.2.

Il s'agissait dans cette section de montrer par l'exemple comment un mécanisme de décision arbitraire peut se connecter à notre système de contrôle, en se conformant à son interface et en profitant des fonctions et structures générées. Pour la suite, concernant la simulation du contrôle et de la décision, bien que le mécanisme à base de contrôle PI soit disponible en tant que fonction générée, nous garderons le mécanisme par défaut choisissant une configuration aléatoirement parmi un espace de configurations accessibles afin de rester focalisé sur notre contribution de contrôle. Le lecteur intéressé par la notion de contrôle PI pourra se référer à [31], qui présente notamment son implémentation et sa

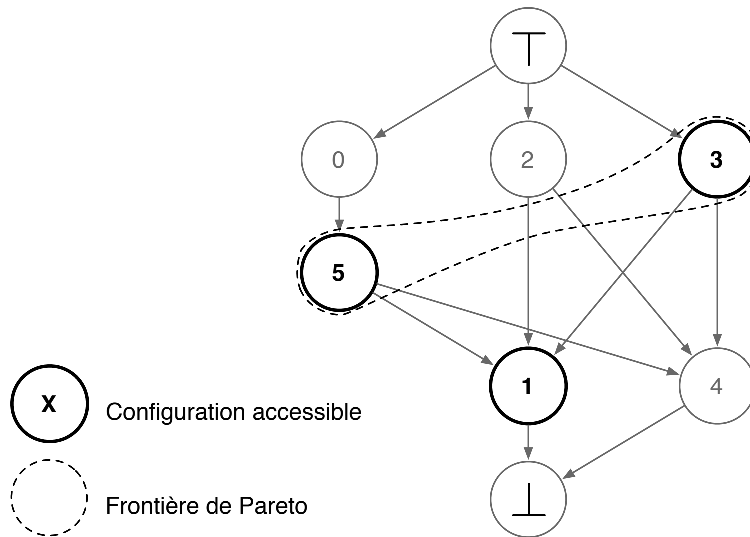


Figure 9.2 – Calcul de la frontière de Pareto dynamique.

simulation sur un espace fini de configurations, en tirant parti du principe d'optimisation Pareto [37] et du vote de Borda pour l'optimisation multi-critères.

9.4 Support pour la simulation

Bien que le code contrôle ainsi préparé soit destiné à être intégré sur une plateforme reconfigurable, il peut être fastidieux de déployer l'ensemble du système lorsqu'il s'agit simplement de simuler la partie contrôle. Il s'agit ici d'encapsuler ce code afin d'en permettre une exécution autonome purement logicielle.

L'objectif est de tirer parti d'outils existants permettant de tracer l'évolution de signaux, en communiquant avec eux via leur protocole, mais également de montrer l'intégration générique avec d'autres outils pour l'envoi d'événements. En effet, notre approche de contrôle peut tout à fait être réutilisée dans d'autre cadres que les architectures reconfigurables, puisqu'elle s'adresse plus génériquement à tout type de système dont l'exécution repose sur le principe des systèmes réactifs synchrones.

37. La seule différence par rapport à notre manière de l'intégrer dans le système de décision étant que cette optimisation est réalisée statiquement une fois pour toutes au lieu d'être effectuée dynamiquement à chaque pas.

9.4.1 Communication vers le contrôleur

L'idée est ici de permettre l'envoi d'événements en entrée du contrôle, et l'émission d'événements via des protocoles exploitables pour le tracé d'exécution. Nous définissons trois manières d'envoyer des événements au contrôleur en mode simulation :

- manuellement en ligne de commande : il s'agit de valuer les événements d'entrée au clavier afin de déclencher des pas d'exécution ; À chaque pas d'exécution, le contrôleur attend que toutes les valeurs d'entrées soient données ;
- automatiquement en ligne de commande : les événements sont préparés statiquement dans des fichiers fournis en paramètre au lancement du contrôleur qui exécute alors le nombre de pas nécessaires pour tous les consommer ;
- manuellement via une interface graphique : le principe est double, il s'agit à la fois de permettre l'intégration du contrôle en tant que brique logicielle dans un contexte autre que les architectures reconfigurables, mais également de simplifier la démarche de simulation en disposant d'une interface graphique adéquate pour cela ; le principe de communication est simple, il s'agira de communiquer avec le contrôleur via une redirection de sortie standard, afin de simuler ce qui se produirait si les entrées étaient fournies au clavier.

Un pas de contrôle s'exécutant en deux temps – un appel au superviseur puis un appel à la décision – il peut également être intéressant de débrancher le mécanisme de décision afin de la maîtriser manuellement lors de l'exécution. Avec un paramètre de lancement adéquat, il est possible d'exécuter le contrôleur de manière à ce qu'il *s'arrête* au moment d'effectuer la décision afin d'attendre la réception d'événements du choix de configuration donnés manuellement, au lieu d'exécuter la fonction de décision implémentée par le concepteur. Le principe est donc d'afficher les configurations permises soit en texte pour une exécution en ligne de commande, soit dans une liste déroulante ce qui est pris en charge par notre outil de simulation graphique (dont une capture d'écran sera donnée lors de la phase de simulation).

9.4.2 Visualisation de l'exécution du contrôle

La visualisation des pas de contrôle consiste à réceptionner des événements (émis par le contrôleur) par un outil spécialisé pour tracer l'évolution de signaux. Ces signaux comprennent dans notre cas la configuration à activer et les valeurs de poids de cette configuration, ainsi que les valeurs des hypothèses et objectifs de contrôle (qui doivent rester à *true* tant que les hypothèses sont respectées).

Deux outils se distinguent à cet effet, Sim2Chro [129] et GTKWave [53], dont l'avantage est d'être capables de fonctionner en temps réel : au lieu de traiter statiquement des fichiers

de traces d'exécutions (ce qu'ils peuvent faire cependant), ils sont capables de fonctionner dans un mode passif, par lequel ils attendent de recevoir des événements pour mettre leur interface à jour.

Sim2Chro

Sim2Chro est un de nos outils de prédilection pour la simulation de systèmes réactifs synchrones, il est notamment intégré à la simulation des programmes en Lustre et des programmes par défaut construits par le compilateur BZR. Puisque nous ne nous servons pas de l'interface par défaut générée par le compilateur BZR dans le démonstrateur (dû à l'ajout du système de décision et au mode d'exécution en deux temps pour un pas) il a fallu adapter l'interface générée afin de produire des événements au format *Reactive Input Format* (RIF) [118] interprétables en temps réel par Sim2Chro.

GTKWave

GTKWave est une visionneuse de formes d'ondes capable d'afficher l'évolution de signaux sous réserve d'être spécifiés dans un format compatible. Cet outil est particulièrement utilisé dans le cadre de la visualisation lors de la simulation de circuits. Il est notamment capable de traiter des événements fournis au format standard *Value Change Dump* (VCD) [38] soit sous forme de fichier (résultat de simulation) soit de manière interactive via un mécanisme de communication à base de blocs de mémoires partagées. Cette deuxième manière nous intéresse plus particulièrement, afin de permettre comme pour Sim2Chro de simuler le contrôleur en ayant un rendu visuel temps réel de son comportement.

La prise en compte des formats RIF et VCD est intégrée dans la couche générée encapsulant le code de contrôle/décision. En lançant le contrôleur avec un paramètre adéquat, il est possible de choisir vers quel format produire les événements de sortie. Il est ainsi possible de connecter le contrôleur au sein d'un mécanisme de communication amont-aval, c'est-à-dire entre deux types d'outils : l'un communiquant des événements et jouant ainsi le rôle de l'environnement (en ligne de commande ou via interface graphique), l'autre réceptionnant les événements produits par le contrôleur afin d'en donner un aperçu visuel, le tout en temps réel.

Un exemple d'utilisation de cette chaîne de communication sera donné dans la section suivante, traitant de la simulation du démonstrateur.

38. `http://en.wikipedia.org/wiki/Value_Change_Dump`

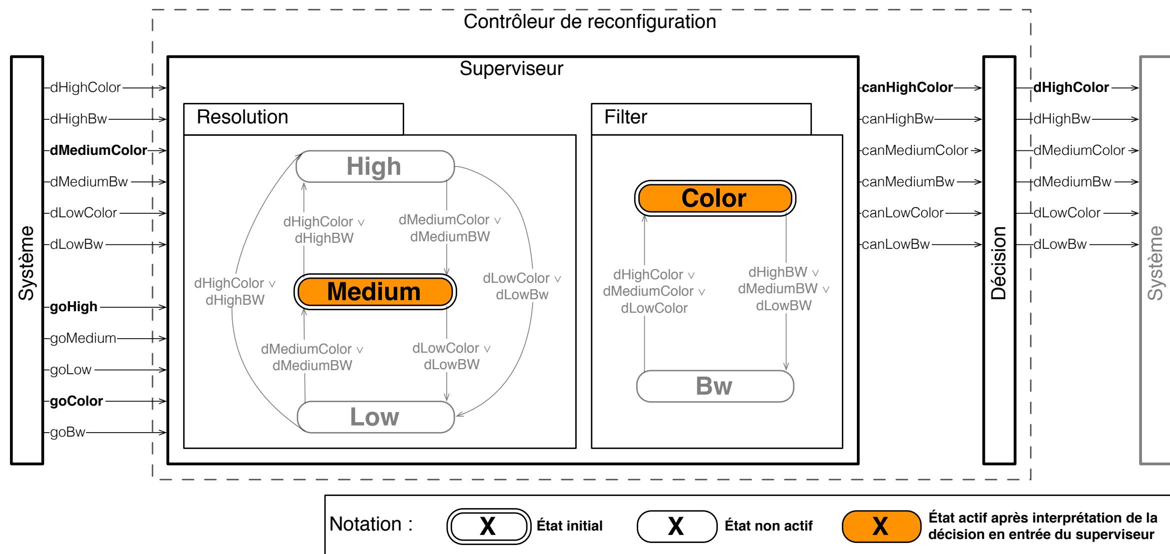


FIGURE 9.3 – Premier pas de contrôle/décision.

9.5 Exécution du démonstrateur

Il s'agit dans cette section de présenter les détails d'exécution du contrôle grâce à la chaîne d'outils de simulation, avant de montrer son intégration finale dans la plateforme reconfigurable.

9.5.1 Simulation

Nous allons ici observer les premiers pas d'exécution du système, qui vont montrer des aspects importants de réaction du contrôleur (superviseur et système de décision).

Premier pas

La figure 9.3 présente le pas initial, pour lequel aucun pas de contrôle n'a encore été effectué et donc aucune décision n'a été calculée pour être envoyée au superviseur. Le système produit des événements non contrôlables vers le superviseur, parmi eux :

```
goHigh ← true
goMedium ← false
goLow ← false
goColor ← true
goBw ← false
```

Il produit également une information de décision initiale, correspondant à sa propre configuration de départ, où seul `dMediumColor` est *true* ce qui coïncide également avec l'état initial du superviseur. Sur occurrence de cette décision initiale, les automates du

superviseur – tels qu'il ont été spécifiés lors des transformations afin de ne réagir que sur la décision – ne modifient pas leur état courant pour ce premier pas (puisqu'ils sont déjà dans leur état initial), et le calcul des prochaines configurations accessibles peut s'effectuer d'après cet état. Les équations données dans les programmes 16 et 17 sont évaluées et seule l'équation suivante est *true* :

$$\texttt{canHighColor} = \underbrace{\texttt{canHigh}}_{\texttt{(medium \& (goHigh \& c)) or ...}} \ \& \ \underbrace{\texttt{canColor}}_{\texttt{...or (color \& not(goBw))}}$$

En effet, le superviseur est pour l'instant en configuration [Medium ;Color], donc les variables reflétant l'activation des états de cette configuration (`medium` et `color`) sont valuées à *true* ; la transition de la garde implicite de l'état `Color` vers lui-même (`Color` $\xrightarrow{\texttt{not(goBw)}}$ `Color`) est également valuée à *true* ; Et enfin, la variable contrôlable c est laissée libre par le contrôleur et prend sa valeur par défaut *true*, autorisant ainsi l'activation de l'état `High`. Le superviseur a en effet pu calculer que la configuration [High ;Color] – accessible en laissant toute variable contrôlable valuée par défaut à *true* – n'enfreint pas la règle de contrôle et ne permet pas non plus de l'enfreindre de manière non contrôlable à partir de cette configuration [39].

Pour ce pas, une seule configuration est autorisée par le superviseur, cependant pour respecter le principe de communication contrôle/décision cette configuration est envoyée au système de décision sous forme d'un espace de configuration, et le système de décision – n'ayant pas d'autre choix légal – renvoie {Medium ;Color} en tant qu'ordre de reconfiguration pour le système.

Deuxième pas

Au pas suivant, le système dispose maintenant d'une information de décision de reconfiguration qu'il convient de communiquer au superviseur, accompagné de nouveau événements, par exemple :

```
goHigh ← true
goMedium ← false
goLow ← false
goColor ← false
goBw ← true
```

En clair, nous allons essayer de rejoindre la configuration {High ; Bw}. L'exécution de ce pas peut être suivi via la figure reffig :CtrlFlow2.

39. Depuis cette configuration, le superviseur dispose toujours de moyens nécessaires pour forcer le système à rester dans un état autorisé pour toute prochaine exécution.

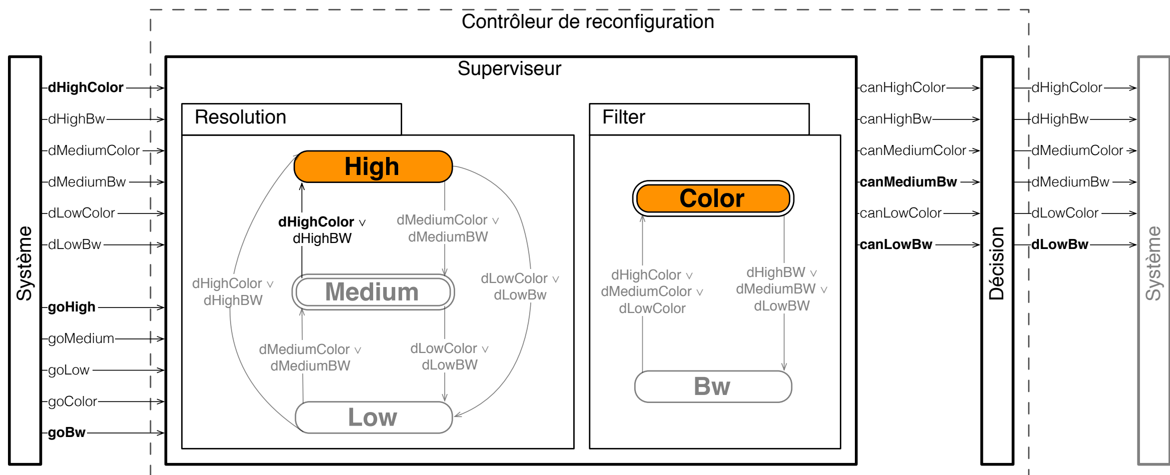


FIGURE 9.4 – Deuxième pas de contrôle/décision.

Le superviseur commence par réagir à la décision du pas précédent, en modifiant l'état de ses automates vers `High` pour celui traitant la résolution, et vers `Color` pour celui traitant du filtre de l'image. Puis il évalue ensuite les accessibilités de configurations. Observons ce qui se produit pour l'accessibilité de la configuration [High ;Bw] :

$$\texttt{canHighBw} = \underbrace{\texttt{canHigh}}_{\texttt{...or (high \& not((goMedium or not(a)) or (goLow or not(a))))}} \;\&\; \underbrace{\texttt{canBw}}_{\texttt{(color \& (goBw)) or ...}}$$

Dans l'évaluation de cette équation pour ce pas, la variable contrôlable a est forcée à *false* par le superviseur, ce qui interdit l'activation de High. Cela signifie que :

– soit {High ;Bw} ne respecte pas l'objectif de contrôle ;
– soit il est possible de rejoindre une configuration interdite via un chemin d'exécution non contrôlable depuis [High ;Bw].

Analysons les propriétés de [High ;Bw], notamment les poids calculés de cette configuration : `power` est évalué à 15 unités, tandis que `qos` est évalué à 13 unités. Pour rappel, l'objectif de contrôle stipule qu'à aucun moment le système ne doit être à la fois sous la barre des 16 unités pour `power` et 14 unités pour `qos`. {High ;Bw} est donc une configuration interdite, c'est pourquoi le contrôleur à valué juste à temps la variable a à faux afin de retirer [High ;Bw] des propositions de configurations.

Mais si cette configuration est interdite, et que la spécification de contrôle a bien pu être synthétisée, c'est donc qu'il existe au moins une configuration alternative, comme précisé dans la partie générique de l'objectif de contrôle.

C'est en effet le cas des configurations [Medium ;Bw] et [Low ;Bw] :

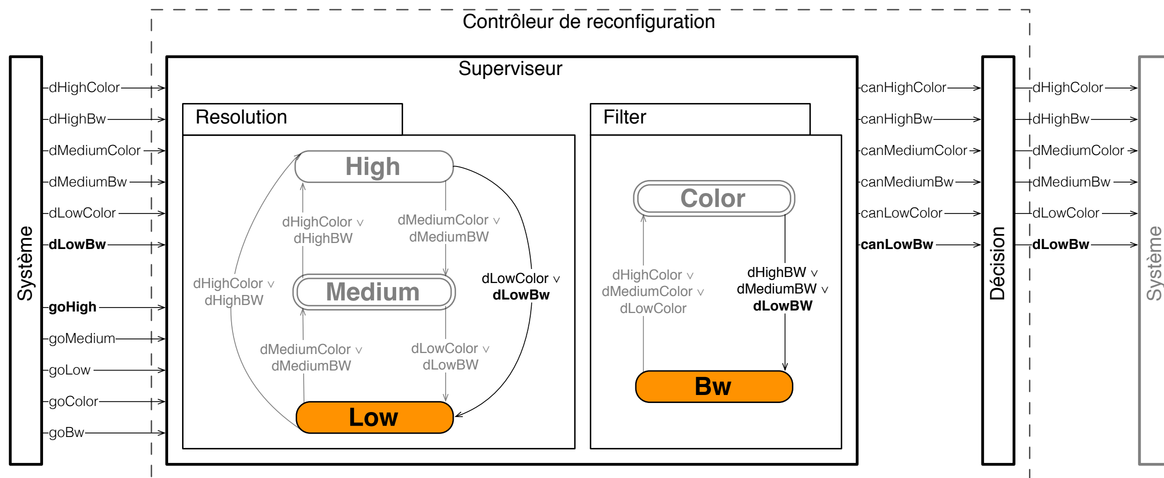


FIGURE 9.5 – Troisième pas de contrôle/décision.

$$\texttt{canMediumBw} = \underbrace{\texttt{canMedium}}_{\texttt{(high \& (goMedium or not(a))) or ...}} \;\&\; \underbrace{\texttt{canBw}}_{\texttt{(color \& (goBw)) or ...}}$$

$$\texttt{canLowBw} = \underbrace{\texttt{canLow}}_{\texttt{(high \& (goLow or not(a))) or ...}} \;\&\; \underbrace{\texttt{canBw}}_{\texttt{(color \& (goBw)) or ...}}$$

La valuation de a à *false* a ici pour effet de bord de rendre ces deux configurations accessibles. Il s'agit d'un cas montrant à la fois la notion de forçage de transition (quelles que soient les valeurs des autre événements, il est toujours possible de forcer le système à quitter `High` et donc à transiter vers les états `Medium` et `Low` depuis `High`) et de proposition de multiples configurations.

Pour le système de décision, il s'agit donc ici de choisir entre [Medium ;Bw] et [Low ;Bw] pour prochaine configuration. Par défaut, nous avons dit que le comportement de la décision est aléatoire, charge au concepteur de l'implémenter de manière plus intelligente. Partons du principe que nous gardons ce système de décision par défaut pour la démonstration, et admettons que la configuration [Low ;Bw] soit choisie. Un ordre adéquat de reconfiguration est alors fourni au système, afin qu'il puisse passer en [Low ;Bw].

Troisième pas

Pour finir dans cette simulation, nous exécutons un troisième et dernier pas, illustré par la figure 9.5. Comme convenu, le système envoie la décision précédente de configuration au superviseur, accompagnée d'une nouvelle valuation des autres variables non contrôlables :

`goHigh` ← `true`

```
goMedium ← false
goLow ← false
goColor ← false
goBw ← false
```

Nous allons ainsi tenter de passer en [High ;Bw] depuis [Medium ;Bw]. Par "tenter", nous entendons le fait que si toute variable contrôlable est libre (donc *true*), alors c'est effectivement le résultat qui doit se produire étant donné la configuration courante et les entrées fournies. Bien entendu, un tel passage à [High ;Bw] doit échouer pour les mêmes raisons qu'évoquées précédemment au sujet de ses poids incompatibles avec l'objectif de contrôle.

Observons l'évaluation de l'accès à {High ;Bw} :

$$\texttt{canHighBw} = \underbrace{\texttt{canHigh}}_{\texttt{(medium \& (goHigh \& c)) or ...}} \;\&\; \underbrace{\texttt{canBw}}_{\texttt{...or (bw \& not(goColor))}}$$

La variable contrôlable c est forcée à *false* par le contrôleur, justement afin d'interdire la transition entre `Low` et `High` puisqu'il n'y a pas moyen de quitter l'état `Bw` d'après la valuation des entrées. Nous avons ici l'illustration de l'autre cas de contrôle nous intéressant : l'interdiction de transiter.

Puisque `High` est inaccessible depuis `Low` dans ce pas, `Low` restera actif et sera proposé combiné à `Bw` en tant que (seule) configuration accessible, retournée ensuite par la décision au système qui cette fois-ci ne se reconfigurera pas puisque l'ordre de reconfiguration correspond déjà à la configuration courante : [Low ;Bw].

9.5.2 Chaîne de simulation

Les trois premiers pas de contrôle/décision qui viennent d'être présentés vont maintenant être présentés via la chaîne de simulation évoquée précédemment. En amont nous lançons un outil graphique capable d'envoyer des événements, y compris de décision, au contrôleur. En aval, Sim2Chro et GTKWave récupèrent les événements produits par le contrôleur, c'est-à-dire la configuration choisie, et les informations de poids.

Les figures 9.6, 9.7, et 9.8 montrent respectivement la production d'événements non contrôlables via une interface graphique – avec une décision manuelle – pour les 3 pas d'exécution qui viennent d'être présentés. À noter que la décision initiale est imposée par le système et n'est donc pas mise à disposition de la simulation manuelle, ainsi le contrôleur est assuré de démarrer correctement en configuration {Medium ;Color}, comme convenu dans la spécification RecoMARTE. Deux remarques sur cette figure :

– l'interface montre le choix de configuration final traduit sous forme d'états activés,

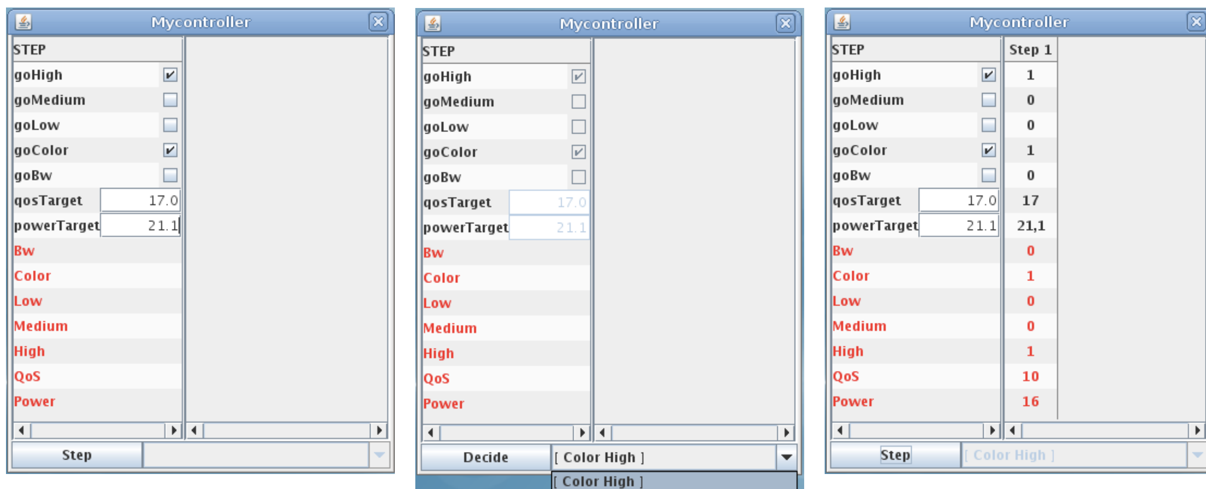


FIGURE 9.6 – Envoi d'événements, contrôle et décision pour le premier pas.

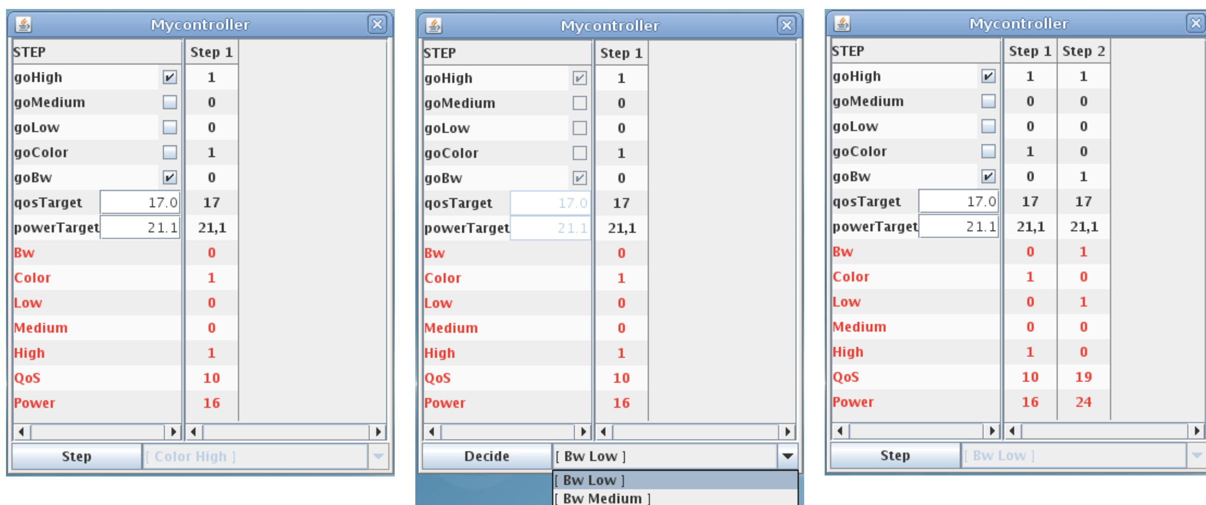


FIGURE 9.7 – Envoi d'événements, contrôle et décision entre deux configurations pour le deuxième pas.

et non sous forme d'une combinaison, celle-ci étant traduite en états actifs dans le code d'encapsulation du contrôleur afin de gagner de la place sur le rendu graphique. (C'est important lorsque les configurations sont nombreuses). Cette remarque sera valable pour Sim2Chro et GTKWave ;

– les événements spécifiques à la décision sont présents et doivent être fournis comme convenu de manière synchronisée avec les autres événements en entrée pour un pas donné, mais ne serviront pas ici puisque nous employons une méthode manuelle afin de maîtriser la décision.

Les deux listings suivant montrent les traces textuelles d'exécution du contrôleur, respectivement au format RIF et VCD.

Mycontroller

STEP		Step 1	Step 2
goHigh	☑	1	1
goMedium	☐	0	0
goLow	☐	0	0
goColor	☐	1	0
goBw	☐	0	1
qosTarget	17.0	17	17
powerTarget	21.1	21,1	21,1
Bw		0	1
Color		1	0
Low		0	1
Medium		0	0
High		1	0
QoS		10	19
Power		16	24

Step [Bw Low]

Mycontroller

STEP		Step 1	Step 2
goHigh	☑	1	1
goMedium	☐	0	0
goLow	☐	0	0
goColor	☐	1	0
goBw	☐	0	1
qosTarget	17.0	17	17
powerTarget	21.1	21,1	21,1
Bw		0	1
Color		1	0
Low		0	1
Medium		0	0
High		1	0
QoS		10	19
Power		16	24

Decide [Bw Low]

[Bw Low]

Mycontroller

STEP		Step 1	Step 2	Step 3
goHigh	☑	1	1	1
goMedium	☐	0	0	0
goLow	☐	0	0	0
goColor	☐	1	0	0
goBw	☐	0	1	0
qosTarget	17.0	17	17	17
powerTarget	21.1	21,1	21,1	21,1
Bw		0	1	1
Color		1	0	0
Low		0	1	1
Medium		0	0	0
High		1	0	0
QoS		10	19	19
Power		16	24	24

Step [Bw Low]

FIGURE 9.8 – Envoi d'événements, contrôle et décision pour le troisième pas.

Production des événements au format RIF :

```
#program Mycontroller
#inputs goHigh:bool goMedium:bool goLow:bool goColor:bool goBw:bool qosTarget:real powerTarget:real
#outputs Bw:bool Color:bool Low:bool Medium:bool High:bool QoS:int Power:int
1 0 0 1 0 17.000000 20.100000 #outs 0 1 0 0 1 10 16
1 0 0 0 1 17.000000 20.100000 #outs 1 0 1 0 0 19 24
1 0 0 0 0 17.000000 20.100000 #outs 1 0 1 0 0 19 24
```

Production des événements au format VCD :

```
$timescale 1 ns $end
$scope module top $end
$var wire 1 ?1 goHigh $end
$var wire 1 ?2 goMedium $end
$var wire 1 ?3 goLow $end
$var wire 1 ?4 goColor $end
$var wire 1 ?5 goBw $end
$var real 64 ?6 qosTarget $end
$var real 64 ?7 powerTarget $end
$var wire 1 !1 Bw $end
$var wire 1 !2 Color $end
$var wire 1 !3 Low $end
$var wire 1 !4 Medium $end
$var wire 1 !5 High $end
$var real 64 !6 QoS $end
$var real 64 !7 Power $end
$upscope $end
$enddefinitions $end
#0
1?1
0?2
0?3
1?4
0?5
r17 ?6
r20,1 ?7
0!1
```

```
1!2
0!3
0!4
1!5
r10 !6
r16 !7
#1
1?1
0?2
0?3
0?4
1?5
r17 ?6
r20,1 ?7
1!1
0!2
1!3
0!4
0!5
r19 !6
r24 !7
#2
1?1
0?2
0?3
0?4
0?5
r17 ?6
r20,1 ?7
1!1
0!2
1!3
0!4
0!5
r19 !6
r24 !7
#3
```

Ces deux traces sont redirigées respectivement en temps réel vers Sim2Chro (par redirection d'entrée standard) et GTKWave (par mémoire partagée). Le format de communication des événements produits par le contrôleur doit être décidé au lancement de celui-ci, la simulation graphique à donc dû être exécutée deux fois pour obtenir les traces sur ces deux outils.

Enfin les figures 9.9 et 9.10 présentent visuellement les traces d'exécution de ces trois pas dans Sim2Chro et GTKWave, correspondant bien au comportement de contrôle que nous avons détaillé précédemment.

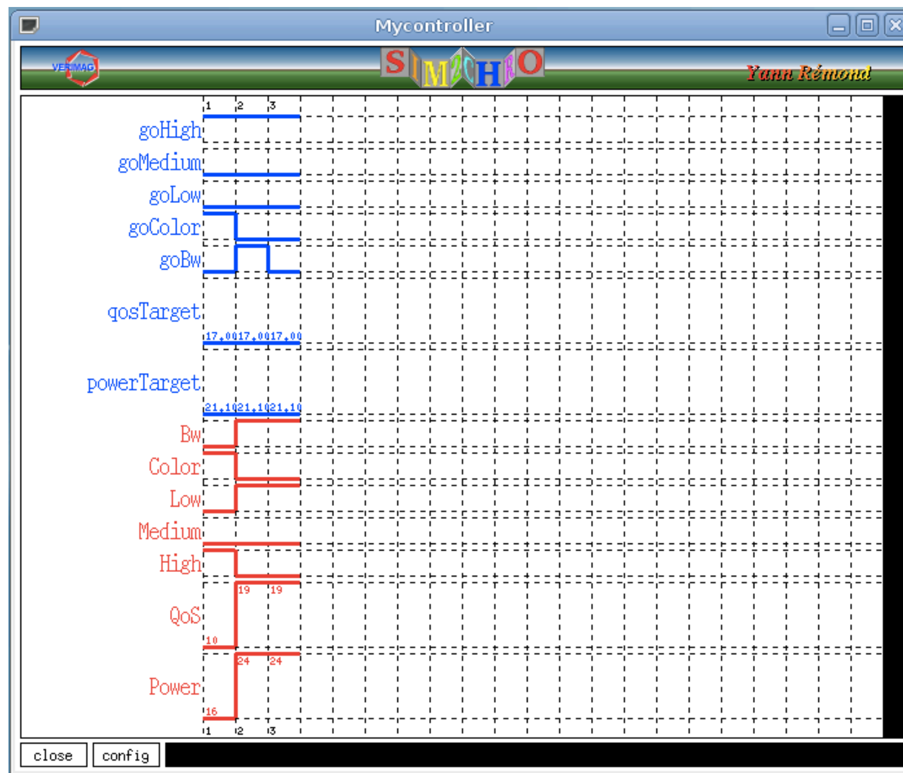


FIGURE 9.9 – Résultat de simulation du contrôleur dans Sim2Chro.

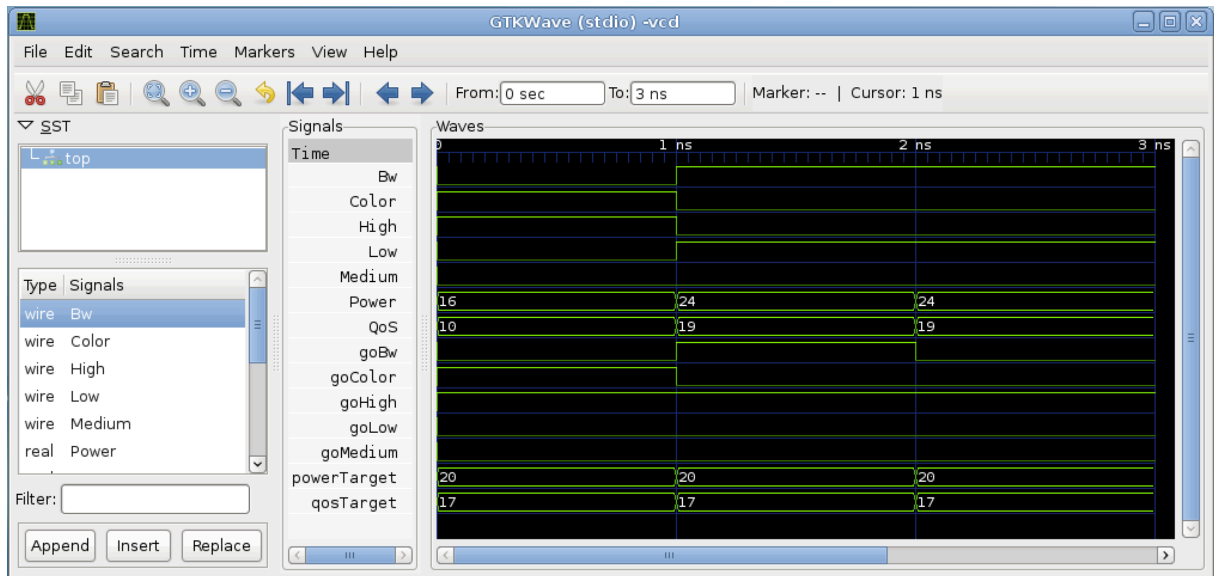


FIGURE 9.10 – Résultat de simulation du contrôleur dans GTKWave.

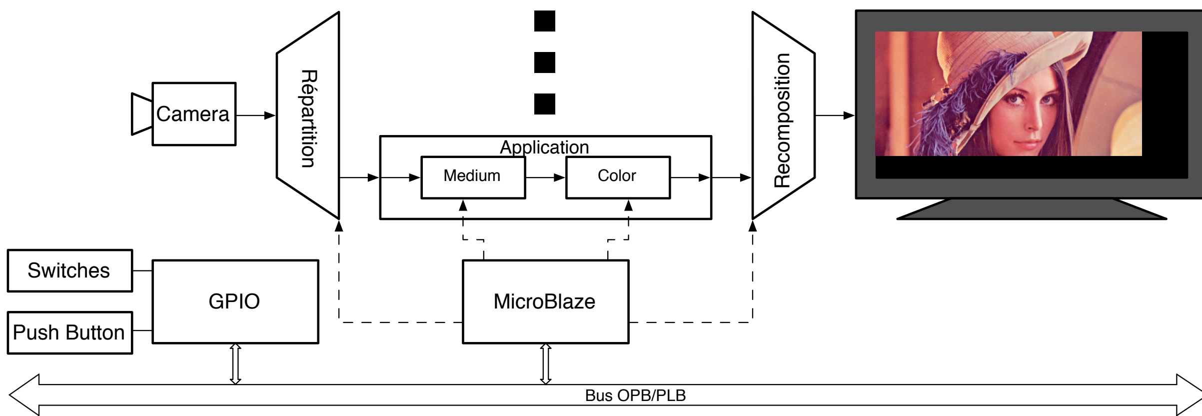


FIGURE 9.11 – Initialisation du système, démarrage en configuration {Medium ;Color}.

9.5.3 Intégration dans la plateforme

L'objectif final de ce contrôleur est ici d'être intégré à la plateforme de démonstration afin de pouvoir la piloter. Pour cela il suffit de ne garder que les fonctions du superviseur et de la décision, et d'écarter le code de simulation qui les encapsule.

Chaque contrôleur (ici il n'y en a qu'un) est ainsi intégré en tant que fonction dans la partie logicielle de la plateforme. Pour rappel, celle-ci dispose à la base d'une boucle globale d'exécution, au sein de laquelle des événements sont reçu par l'environnement (ici via des switches et un bouton) pour être consommés par le/les contrôleur(s), chacun exécutant alors un pas (contrôle et décision) par itération. À la fin d'une itération les ordres de reconfigurations sont récupérés puis traduits en ordres de bitstreams partiels à charger. Cette opération s'effectue via la fonction `loadbitstream` du framework, qui se charge de placer le bitstream d'un PE donné lorsque son implémentation courante à terminé son traitement sur une image. Il nous suffit simplement de spécifier dans cette boucle un appel à notre contrôleur, en branchant les événements récupérés correspondant à `goHigh, goMedium, goLow, goColor` et `goBw`.

La production d'événements, auparavant effectuée au clavier ou via une interface graphique lors de la simulation, s'effectue ici manuellement via des switches et un bouton : cinq switches sont employés pour valuer les variables d'entrées `goHigh, etc.`, puis lorsqu'une valuation est prête il suffit d'appuyer sur un bouton pour transmettre l'ensemble des valeurs de manière synchrone sur un pas d'exécution du contrôleur.

Les figures 9.11, 9.12, 9.13 et 9.14 montrent les trois pas d'exécution précédents, effectués sur la plateforme (en supposant que la décision aléatoire choisisse à nouveau {Low ;Bw} au deuxième pas).

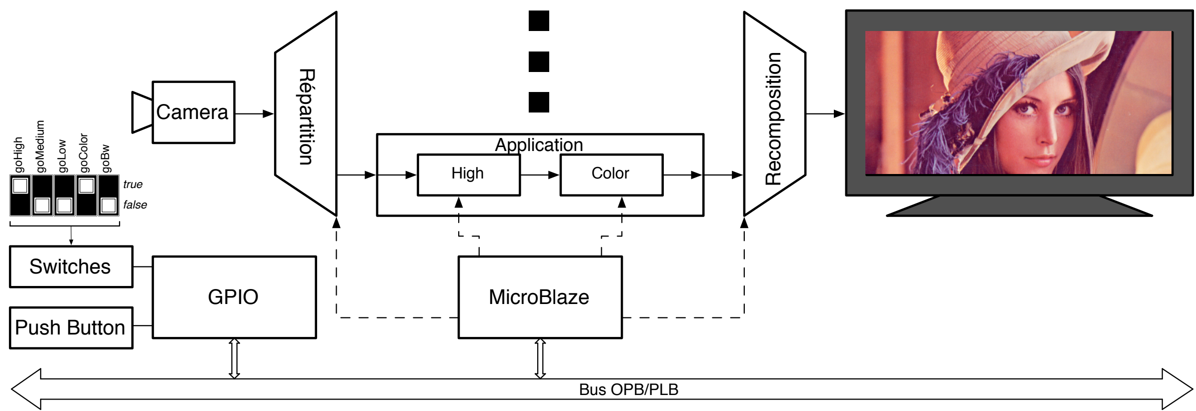


FIGURE 9.12 – Premier pas, reconfiguration vers {High ;Color}.

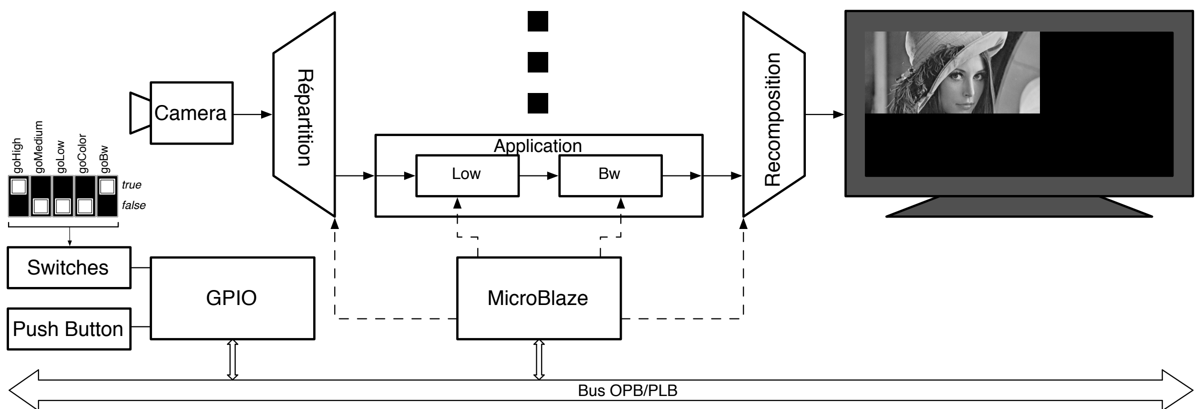


FIGURE 9.13 – Deuxième pas, {High ;Bw} interdite, choix de reconfiguration vers {Low ;Bw}.

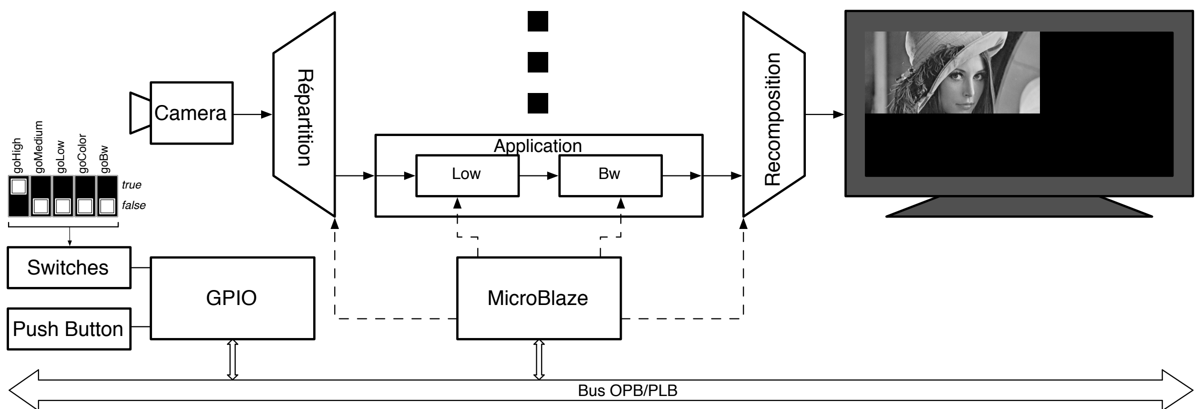


FIGURE 9.14 – Troisième pas, {High ;Bw} interdite, le système reste en {Low ;Bw}.

9.6 Conclusion

Ce chapitre vient clore la partie "validation" de cette étude en montrant l'automatisation du flot des transformations depuis un modèle RecoMARTE jusqu'à la simulation du contrôle de reconfiguration et son intégration sur une plateforme reconfigurable.

Nous avons vu les détails de génération de la spécification synchrone – appliquant les règles de transformation spécifiées sous forme équationnelle au chapitre 5 – contenant le code de contrôle donné par contraintes (spécification comportementale à base d'automate et d'équations, combinée à des hypothèses d'utilisation et des objectifs de contrôle. Ce code est ensuite synthétisé en faisant appel au compilateur BZR, qui lui-même fait appel à la synthèse de contrôleur via SIGALI afin de générer le code complet de la partie contrôle : le superviseur.

Ce superviseur est ensuite combiné à un mécanisme de décision dont la structure est également générée dans le but d'être complétée par le concepteur, si besoin au moyen de fonctions de contrôle générées (contrôle PI dont les coefficients sont à adapter) ou de fonctions et structure fournies pour accéder de manière ordonnée aux configurations (treillis, Pareto) ou encore de fonctions pour l'optimisation multi-critères (vote de Borda).

Avant intégration finale, le concepteur dispose d'une chaîne d'outils lui permettant de simuler son contrôleur (superviseur et décision) de manière purement logicielle, lui évitant ainsi de déployer systématiquement le résultat des transformations vers sa plateforme reconfigurable lorsqu'il s'agit simplement de simuler le contrôle. À noter que cette étape montre précisément l'indépendance de l'exécution vis-à-vis de toute plateforme ; en effet, concernant la méthodologie proposée, seule la partie (Reco)MARTE est dédiée aux architectures. Enfin, l'intégration dans la plateforme est ici simplifiée car nous proposons une plateforme de base, alliée à un framework pour le déploiement d'applications de traitement vidéo, disposant d'une boucle globale d'exécution dans laquelle il est trivial d'implanter les appels aux fonctions de contrôles générées par transformation et de diffuser leurs ordres de reconfiguration via la fonction `loadbitstream`.

Cette démonstration vient de montrer dans son ensemble la méthodologie de conception, transformation, simulation et intégration défendue dans cette étude. L'exemple proposé (application et plateforme) peut à présent être itéré au sein du projet FAMOUS.

Conclusion générale

1 Bilan

L'objectif des travaux présentés ici était de comprendre comment allier la modélisation du comportement de systèmes reconfigurables à la notion de contrôle sûr de reconfiguration, en étant motivé par la mise en œuvre d'un socle de conception sur lequel le projet FAMOUS pourra itérer de manière modulaire. Nous passons ici en revue les principaux points de contribution concernant ce socle.

1.1 Définition d'une méthodologie reposant sur UML/MARTE

La première contribution apportée par cette étude est l'extension de MARTE pour la définition de la partie contrôle d'un système réactif. La proposition permet, au sein d'un modèle d'application, de distinguer des flux d'événements de deux types : le flot de données, soit l'information réservée aux tâches classiques, et le flot de contrôle, c'est-à-dire les événements captés par le système et devant être analysés car impactant son comportement ; cette analyse s'effectue au moyen de tâches identifiées comme réservées au contrôle de reconfiguration. L'avantage étant ici de pouvoir spécifier deux modèles de calcul différents : un pour les tâches, et un pour le contrôle, ce qui permet de spécifier des systèmes recevant des événements de contrôle à des cadences indépendantes du flot de données. L'exemple donné en démonstration illustre d'ailleurs ce cas, puisque d'un côté les images traitées par les PE arrivent à une fréquence régulière, tandis que les événements de contrôle – provoqués par l'utilisateur en positionnant des switches et en appuyant sur un bouton – peuvent arriver à n'importe quel moment.

La méthodologie permet également d'associer des poids à des représentations partielles de configurations (via les Configurations MARTE), ce qui autorise par la suite à spécifier des contraintes non seulement sur les compatibilités (ou incompatibilités) de Configurations MARTE entre elles, mais également des contraintes sur les compositions de poids. Des moyens d'associer des poids existaient au préalable dans MARTE, mais la méthodologie devenait très vite complexe avec l'augmentation du nombre d'états et donc de

compositions d'états, car le concepteur était obligé de calculer et spécifer lui-même les combinaisons de poids (en essayant de ne pas faire d'erreur...). Dans notre approche, ces mécanismes de spécifications de poids ainsi que leur valuation ont été améliorés afin d'automatiser un maximum de calcul. Seuls les poids pour chaque Configuration MARTE sont à définir, les poids des compositions étant automatiquement déduits par la suite.

1.2 Spécification de transformations

Formaliser les moyens de modélisation d'un concept est certes un apport important, mais sans transformation associée, une telle proposition perdrait du potentiel par rapport à l'objectif d'UML dont la vocation est d'aller au-delà de la simple documentation d'un système et de parvenir à une génération de code effectif et intégrable à partir de modèles. C'est le principe d'une approche d'Ingénierie Dirigée par les Modèles.

Des transformations ont donc été formalisées dans cette étude afin de permettre, une fois implémentées, d'obtenir automatiquement du code d'outillage et d'intégration, ainsi qu'une spécification de contrôle, exploitant le langage synchrone BZR ; les programmes BZR générés servant ensuite à produire, par compilation et synthèse, la partie la plus importante d'un contrôleur de reconfiguration : le superviseur.

1.3 Intégration du contrôle discret statique et sûr à la décision pour optimisation

Au sein de la méthodologie, deux entités dédiées à la maîtrise des ordres de reconfiguration cohabitent : le superviseur, capable de restreindre les possibilités d'évolutions du système de manière sûre, et la décision, dont le rôle est de choisir une évolution parmi celles qui sont possibles.

L'intérêt principal de la méthodologie est de tirer parti de mécanismes de synthèse de contrôleur, afin de pouvoir spécifier le contrôle par contraintes – au lieu de nécessiter son implémentation manuelle – et d'obtenir son implémentation finale (si elle existe) automatiquement, et ce de manière formelle.

En observant le comportement général des contrôleurs générés, nous nous sommes aperçu qu'un besoin en terme d'optimisation n'était pas encore traité. En effet, bien que des algorithmes existent pour optimiser statiquement le contrôleur sur une fenêtre de temps [40], cela n'était pas le cas pour l'optimisation dynamique.

Ainsi, en modifiant la spécification de contrôle afin que le superviseur calcule non plus une prochaine configuration à mettre en œuvre mais un espace de configurations légales, il

40. Algorithmes de *synthèse optimale de contrôleur*, précalculant les valeurs optimales des variables contrôlables

est devenu intéressant de séparer ce qui relève purement de la sécurité de reconfiguration (le contrôle) de ce qui permet de l'optimiser (choix parmi plusieurs configurations légales).

C'est pourquoi nous avons adapté nos transformations afin de permettre la génération de la partie structurelle d'un mécanisme de décision, implémentable par le concepteur, au moyen de fonctions également générées à cet effet. La spécification de contrôle (BZR) a également été adaptée afin de prendre en compte de manière formelle l'externalisation d'un partie (le principe de décision) qui était auparavant sous sa responsabilité.

1.4 Proposition d'une plateforme reconfigurable

Un des besoins majeurs du projet FAMOUS est la définition d'un support d'exécution potentiellement capable de démontrer l'ensemble de ses modules, c'est-à-dire l'ensemble des contributions apportées par les membres du projet.

La proposition d'une plateforme reconfigurable fait partie des contributions de cette étude. Nous avons en effet pu exposer les besoins relatifs au projet, afin de justifier les choix pour la définition de la plateforme :

- elle se pose en exemple de ce qui devrait être généré à terme via le flot complet de modélisation FAMOUS, comprenant notre partie contrôle, mais également le modèle complet d'application, d'architecture, et d'allocation, en passant par l'incorporation d'IP tierces via IP-XACT ;
- elle permet la mise en œuvre d'applications multi-tâches reconfigurables, exploitables pour notre démonstration de contrôle mais également pour des mécanismes de contrôle plus fins – proposés par un autre acteur du projet – capables de d'influencer le flot de données (séquencement de tâches) ;
- enfin elle propose un framework bas niveau simplifiant l'exécution de tâches et la mise en place des configurations, ce qui servira de brique de base au mécanisme de diffusion des ordres reconfigurations du projet FAMOUS (membrane).

1.5 Mise en place d'une chaîne de simulation

Enfin, bien que la partie *contrôle* dans notre approche soit formelle par rapport à des contraintes imposées statiquement [41], la partie décision reste une étape dans le contrôleur de reconfiguration qu'il peut être intéressant de simuler, par exemple afin d'observer que son comportement tend bien vers une optimisation en général.

Pour cette raison, afin d'être capables de montrer une intégration du contrôleur hors du contexte des architectures reconfigurables, nous avons proposé un outillage du code

41. Ce qui affranchit donc de la nécessité de valider ou de tester par simulation le code de contrôle obtenu par synthèse afin de s'assurer de son comportement correct.

généré, afin de communiquer avec le contrôleur et récupérer ses traces d'exécution via des outils d'affichage d'ondes existant (Sim2Chro et GTKWave). Un outil de simulation graphique en amont du contrôle (envoi d'événements) a également été proposé, permettant de montrer et de prendre la main sur les étapes de contrôle et de décision intervenant dans un pas global de contrôle.

1.6 Comparaison par rapport aux travaux fondateurs

Cette étude a hérité de travaux majeurs, spécialisés chacun dans la modélisation, le contrôle ou la décision de reconfiguration. En plus des contributions qui viennent d'être résumées, voici de manière synthétique les avancées spécifiques de nos travaux comparées à leurs apports originaux :

- **Comparé à MoPCoM [136]**, approche spécialisée dans la modélisation de systèmes dynamiquement reconfigurables avec UML, nous proposons la séparation des flots de contrôle et de données, et plus généralement la notion même de contrôle est séparée de la diffusion d'une reconfiguration dans FAMOUS ; l'implémentation du contrôle seul est alors beaucoup moins contraignante dans cette approche, puisque qu'elle peut être effectuée indépendamment des tâches dédiées au traitement des données.
- **Comparé à l'approche Gaspard [111]**, méthodologie de modélisation de systèmes reconfigurables basée sur MARTE avec extensions, nous nous tenons à la définition officielle des modes dans MARTE qui n'est pas réduite à la simple expression *un mode = une implémentation d'une tâche* ; nous décorrélons d'ailleurs les définitions structurelles et comportementales des tâches, afin de permettre la représentation d'aspects plus poussés de configurations (une Configuration MARTE, connectée à un mode, sert à définir partiellement une configuration du système). De plus nous avons une portée de modélisation plus générique, non restreinte au modèle de calcul Array-OL pour l'exécution des tâches traitant le flot de données.
- **Comparé aux travaux présentés dans [144]**, axés sur la vérification formelle de comportement de reconfiguration dans les systèmes dynamiquement reconfigurables et soulevant l'intérêt de la synthèse de contrôleur en montrant un exemple d'utilisation manuel, nous avons une proposition effective d'intégration de la synthèse de contrôleur dans notre méthodologie. Celle-ci est d'ailleurs effectuée de manière transparente pour le concepteur. En effet, à aucun moment il ne lui est nécessaire de connaître l'outil de vérification et synthèse de contrôleur Sigali, ou de connaître le langage BZR (les programmes BZR générés constituent une étape intermédiaire avant l'obtention d'un code de contrôle, cette fois-ci en C) ;
- **Enfin, comparé à l'étude [31]**, qui propose un mécanisme d'optimisation de

reconfiguration par contrôle proportionnel et intégral pour la gestion de reconfiguration dans les systèmes dynamiquement reconfigurables, nous avons une approche permettant non seulement d'être correct (toutes les configurations sont supposées légales dans [31]), mais ayant en plus la capacité d'incorporer ce mécanisme d'optimisation en tant que système de décision après contrôle.

2 Perspectives

Nous allons ici distinguer deux types de perspectives concernant ces travaux : les améliorations potentielles de notre méthodologie, et l'intégration dans le projet FAMOUS.

2.1 Perspectives sur la méthodologie proposée

Reconfiguration du contrôleur

Les raisons conduisant au besoin de reconfiguration du contrôleur sont multiples. D'une manière générale dans notre approche, un système dynamique piloté par un contrôleur est relativement figé : son espace de configurations est fini et fixe, ses poids sont définis statiquement, etc. Une mise à jour du système implique souvent la modification appropriée du contrôle, il peut s'agir : de corriger le système, de prendre en compte de nouvelles implémentations, de nouveaux poids, de modifier structurellement le comportement du système (changement du système de transitions, suppression/ajout/modification d'un état), de faire évoluer les objectifs de contrôle, etc.

Bien sûr, cela n'est pas un problème si pour une quelconque modification du contrôle, ou modification du système impactant le contrôle, le système est interrompu – dans le but d'effectuer les modifications nécessaires – puis réinitialisé. Mais si nous tenons à garder un principe de reconfiguration *dynamique*, alors de nouveau enjeux sont à prendre en compte. Comment prévoir les évolutions structurelles du système ? À défaut, comment définir un cadre de liberté, dans lequel un contrôleur peut s'insérer, puis laisser sa place à un autre ? Comment synchroniser le passage d'un état de contrôle à un autre appartenant à un contrôleur différent ? Comment assurer qu'un contrôleur est toujours reconfigurable ?

Telles sont les premières questions que nous devons nous poser afin de faire évoluer la méthodologie proposée dans le but de prendre en compte la reconfiguration du contrôle.

Un premier élément de réponse nous apparait pour la dernière question : dans notre approche nous n'avons utilisé qu'un sous ensemble des possibilités de synthèse de contrôleur offertes par SIGALI, lui-même exploité par BZR, plus particulièrement nous avons

tiré profit de la propriété d'*invariance*, vérifiable par synthèse via SIGALI [42] ; cependant il existe d'autres propriétés, en particulier la notion d'*accessibilité*.

Un ensemble d'états est dit *accessible* depuis un état y si, pour tout état x de cet ensemble, il existe une trajectoire contrôlable depuis l'état y menant à x. Une première piste à explorer serait donc ici d'employer SIGALI pour assurer qu'un espace de configurations définies comme stables pour la modification du contrôleur est toujours accessible depuis tout état. L'objectif étant de prévoir et d'assurer l'accessibilité d'un état de contrôle dans lequel le contrôleur peut être reconfiguré par un autre, ce dernier démarrant alors à son tour depuis un état stable.

Coût de reconfiguration

L'application de poids dans notre méthodologie a été montrée sur les Configurations MARTE. La combinaison des poids à été interprétée comme le coût statique payé à chaque pas de contrôle : si une configuration du système dispose d'un poids w évalué à 10 unités, ces 10 unités sont payées à chaque pas sans variation dès son activation. Or dans les architectures reconfigurables, le processus de reconfiguration peut lui-même avoir un coût (temps, consommation d'énergie, etc...). C'est d'ailleurs le cas dans notre plateforme : le changement d'un bitstream n'est ni instantané, ni gratuit en terme de consommation. Bien qu'il soit possible de représenter une telle variation de poids de configuration dans notre approche [43] – et nous allons voir un exemple – cela peut vite s'avérer fastidieux avec l'augmentation du nombre d'états. En effet, la prise en compte des coûts de reconfiguration implique la création de nouveaux états et transitions capables de simuler le fait de rester où non dans un configuration.

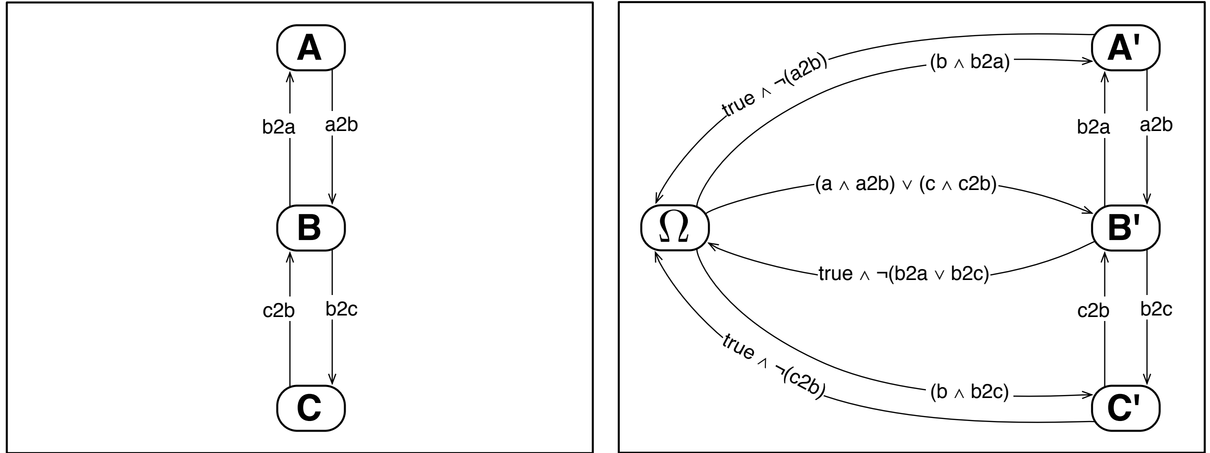
Le principe proposé est le suivant, cf figures 1a et 1b : pour tout automate A de la spécification RecoMARTE sans prise en compte des coûts de reconfiguration, construire un automate A' disposant des mêmes états et transitions auquel on rajoute un nouvel état (nommé Ω dans la figure).

Ensuite, soit b_q une variable reflétant l'activation de l'état q dans A et soient q et q' deux états de A et $\hat{q}$ et $\hat{q}'$ leur état correspondant dans A', connecter tout état de A' à son état Ω par une transition. Les gardes g de ces transitions sont définies de cette manière :

$$\forall(\hat{q} \xrightarrow{g} \Omega) \in A', g = \neg(\{\bigvee g' \mid g' \in (q \xrightarrow{g'} q'), q \neq q'\})$$

42. Le fait d'assurer que le système contrôlé restera toujours dans un espace d'états autorisés.

43. Le poids d'une configuration varie suivant si elle est déjà instanciée (coût statique) ou si elle vient d'être instanciée (coût statique + coût de reconfiguration)



(a) Automate d'origine.

(b) Nouvel automate correspondant, composé avec l'automate d'origine.

FIGURE 1 – Méthode systématique de modélisation permettant ensuite de simuler des coûts de transition.

Ainsi lorsque l'on reste plus d'un pas dans un état de A cela se traduit par une transition de l'état correspondant vers Ω dans A'.

Connecter ensuite réciproquement l'état Ω de chaque nouvel automate à tout état $\hat{q}'$ par une transition, tel que les gardes g de ces transition s'expriment ainsi :

$$\forall (\Omega \xrightarrow{g} \hat{q}') \in A', g = (\{\bigvee (b_q \wedge g') \mid g' \in (q \xrightarrow{g'} q'), q \neq q'\}$$

De cette manière, lorsque l'on sort d'un état q dans A dans lequel on était depuis au moins deux pas pour aller vers un état q', cela se traduit par une sortie vers $\hat{q}'$ dans A' car b_q est *true* (c'est l'état courant) et la garde g' de la transition de q vers q' est *true* également. (À noter que le cas d'une sortie de q alors que q était seulement actif au pas précédent est trivial puisqu'il s'agit d'emprunter la transition correspondante dans A', le système de transition sans Ω étant identique pour A et A').

Enfin dans RecoMARTE, associer chaque nouvel état (mode d'un ModeBehavior) à une Configuration MARTE via laquelle on peut alors rattacher un poids qui correspondra à un poids de transition vers cet état. Et bien sûr, pour chaque Configuration MARTE liée à un état Ω, spécifier des valeurs de poids neutres par rapport à leur opération de combinaison (exemple 0 dans le cas de l'addition). De cette manière lorsqu'un état Ω est actif, ses poids ne s'ajoutent pas (ne modifient pas) aux autres poids des Configurations MARTE actives.

On passe ainsi d'un besoin de variation en terme de poids à une représentation sans

variation puisque les véritables configurations du système sont alors les combinaisons des états de tous les automates. Le poids du système à un instant donné – incorporant le poids statique et l'éventuel poids des transitions venant d'être effectuées – est donc bien représenté par la combinaison des poids de toute Configuration MARTE dont l'état (de A ou A') associé est actif.

Si la méthode proposée permet bien de modéliser la notion de coût de transition [44], de manière exploitable sans modification des transformations spécifiées dans cette étude pour la génération du contrôle, un inconvénient majeur est à relever : de toute évidence il n'est pas efficace de laisser le concepteur réaliser une tâche aussi systématique. En effet, non seulement une telle approche devient vite impraticable manuellement (du fait de l'explosion combinatoire), et comporte beaucoup de redondances (duplication d'états et transitions), mais en plus la mise en œuvre des A' pourrait être automatisée en suivant le principe qui vient d'être décrit.

L'automatisation aurait d'ailleurs deux autres avantages en plus d'alléger la charge de travail pour le concepteur :

- Le masquage des variables reflétant l'activation des états : la manière proposée de spécifier le coût des transitions fait appel au nommage de variables b_q qui certes ont une existence via les transformations finales vers BZR, mais ne sont déclarées nulle part dans la spécification RecoMARTE (elles ne sont pas données en entrées des automates, puisqu'elles sont sensées être internes) ce qui est inconsistant.
- La non ambiguïté des transitions : en effet d'après la figure 1b, si depuis Ω nous avons $b = true$ ainsi que $b2a = true$ et $b2c = true$, il y a alors ambiguïté du fait du non déterminisme de la spécification RecoMARTE (il est possible de rejoindre A' comme C'). Le concepteur est alors responsable du fait de synchroniser correctement les automates A et A' afin que l'activation d'un état q dans A corresponde bien soit à l'activation de Ω dans A' (si l'on ne transite pas) soit à une activation de l'état correspondant $\hat{q}$ dans A'. Cette ambiguïté serait automatiquement levée si les automates A étaient ceux déjà transformés dans la spécification BZR. En effet les transitions de A' seraient implicitement non ambigües puisque la décision (sur laquelle réagissent les automates transformés A) ne spécifie qu'un et un seul booléen à *true* à chaque pas (provoquant ainsi l'activation d'une et une seule transition par automate transformé).

Afin de mettre en œuvre une telle automatisation, il serait nécessaire au préalable de proposer une amélioration de la méthodologie de modélisation (modification du métamo-

44. Où plus précisément le coût d'activation d'un état. On pourrait aller plus loin en proposant le coût de sortie d'un état, reposant sur un principe de modélisation similaire.

dèle RecoMARTE) capable de représenter de manière simple et sans redondance (sans duplication d'états ou transitions) la notion de coût de reconfiguration.

Contrôle en pipeline

Lors d'une reconfiguration d'un système impactant plusieurs de ses tâches, le processus de diffusion de l'ordre suit le modèle de calcul défini (dans notre cas simple, on reconfigure les tâches sources de notre graphe de tâches lorsqu'elles ont terminé leur traitement, puis l'ordre est diffusé à leurs éventuelles tâches filles et ainsi de suite jusqu'à ce que le graphe soit complètement traité).

Il se produit alors un phénomène intéressant : pendant le processus de reconfiguration une partie du système est dans une configuration courante alors que l'autre partie est déjà reconfigurée. Ces deux parties s'exécutent ainsi en concurrence – traitant deux séquences distinctes dans le flot de données – et à un instant physique précis considéré pendant la reconfiguration, le système se trouve dans une configuration intermédiaire.

L'approche courante ne traite pas ces cas de configurations intermédiaires car elle ne considère que des configurations logiques. Or nous nous pourrions ici imaginer un cas d'application dans lequel le passage par une configuration intermédiaire est problématique, occasionnant par exemple un surcoût de consommation, allant à l'encontre d'éventuels objectifs de contrôle. On pourrait éviter simplement ce phénomène en suspendant systématiquement l'exécution du système à la fin d'une séquence donnée afin de le reconfigurer complètement puis de reprendre ensuite l'exécution, mais cela n'est pas forcément la meilleure solution (surtout d'un point de vue temps de traitement) et l'idée ici est de pouvoir diffuser une reconfiguration en pipeline.

De plus, pour un ordre de reconfiguration donné, plusieurs méthodes de diffusion peuvent exister. Prenons l'exemple d'un système composé de deux tâches s'exécutant en parallèle et disposant chacune de deux implémentations possibles, respectivement $T1$, $T1'$ et $T2$, $T2'$. Partons du principe que le système est en configuration [T1,T2] à un instant logique t, et qu'un ordre de reconfiguration est reçu pour passer en configuration [T1',T2'] à l'instant logique $t + 1$.

L'ordre de reconfiguration peut alors être parallélisé, et si $T1$ est reconfiguré vers $T1'$ alors que $T2$ n'a pas encore fini son traitement pour la séquence courante, alors dès l'activation de $T1'$ le système est en configuration intermédiaire [T1' ;T2]. Même chose réciproquement si $T2$ est reconfiguré vers $T2'$ avant $T1$.

Définissons maintenant un poids w, composable par addition, et associons des valeurs à ce poids pour chaque implémentation, tel que $T1 : w = 5$, $T2 : w = 3$, $T1' : w = 2$, $T2' : w = 8$, et supposons qu'il fasse partie des objectifs de contrôle de garantir que $w < 12$ pour toute exécution. [T1 ;T2] et [T1' ;T2'], de poids respectifs calculés 8 et 10

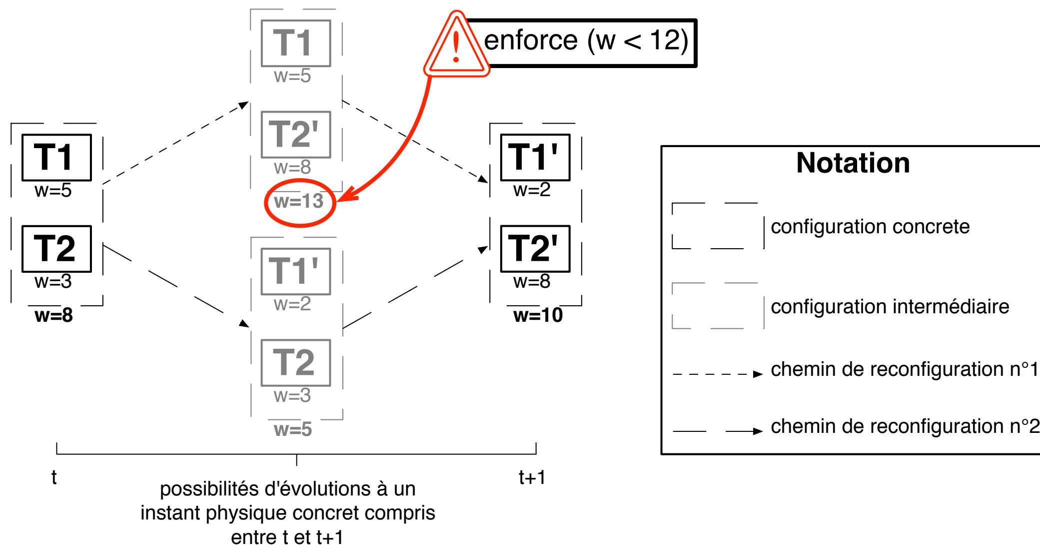


FIGURE 2 – Détails des configurations intermédiaires possibles d'une reconfiguration en pipeline.

pour w, sont techniquement compatibles avec l'objectif de contrôle, et si aucun chemin non contrôlable ne peut mener le système en échec depuis [T1' ;T2'], alors notre approche de contrôle présentée jusqu'ici ne doit pas interdire la transition de [T1 ;T2] vers [T1' ;T2'] si les conditions sont réunies pour la franchir.

Cependant, sur la figure 2, nous pouvons observer que le chemin de reconfiguration n°1, passant par la configuration intermédiaire [T1 ;T2'], n'est pas compatible avec l'objectif de contrôle, tandis que le passage par [T1' ;T2] dans le chemin n°2 permet d'aboutir légalement à la configuration finale [T1' ;T2'].

Nous obtenons ainsi un cas particulier où une diffusion de reconfiguration peut aboutir sans compromettre le système, mais la méthode de diffusion doit par contre être soigneusement sélectionnée afin de ne pas prendre un chemin interdit.

Une telle responsabilité pourrait relever du mécanisme de diffusion de reconfiguration prévu dans FAMOUS, la membrane, mais nous voyons que nous sommes ici face à un problème d'exploration combinatoire, avec des séquences interdites, soit un problème particulièrement intéressant pour la synthèse de contrôleur.

Les configurations intermédiaires devraient pouvoir disposer de leurs propres contraintes, séparées des objectifs de contrôle réservés aux configurations logiques. En effet, par exemple $T1'$ peut très bien être être incompatible avec $T2$ sur une même séquence d'exécution sans toutefois poser de problème de cohabitation lors d'une reconfiguration en pipeline (où $T1'$ et $T2$ traitent alors deux séquences de données différentes). Une contribution possible pour la présente approche consisterait donc à modifier la méthodologie de

modélisation, ainsi que les mécanismes de contrôle associés, afin de prendre en compte les caractéristiques se produisant pendant la diffusion d'un ordre de reconfiguration.

Optimisation de l'intégration contrôle/décision

L'exemple d'algorithme de décision – à base de contrôle PI – qui nous a servi ici de support pour conceptualiser l'intégration contrôle/décision, raisonne sur un espace de configurations explicites : les combinaisons d'états, issues de la composition synchrone des états des automates du système sous contrôle, sont explicitement calculées afin que la décision puisse avoir accès aux valeurs de poids de chacune d'entre elles séparément.

Dans notre approche, cette représentation explicite implique l'insertion d'un nombre exponentiel de variables d'états (correspondant aux configurations). Or Sigali, comme tout outil de synthèse de contrôleur, est très sensible au nombre de variables d'états, ce qui limite donc l'approche à des problèmes de taille peu élevée (la barrière de la centaine de milliers d'états pouvant être rapidement franchie). Cependant, rappelons que malgré cette limitation, les problèmes pris en charge par la solution restent difficiles à résoudre via une éventuelle approche manuelle (puisqu'il s'agit ici d'obtenir automatiquement le contrôle – s'il existe – et de manière maximalement permissive, au lieu d'essayer de le trouver à la main quitte à le vérifier ensuite).

Donc bien que cette solution présente un intérêt vis-à-vis des autres approches, force est de constater que l'optimisation de l'intégration contrôle/décision mériterait d'être approfondie : Sigali, comme la plupart des outils de synthèse de contrôleur modernes, est capable de raisonner sur des représentations symboliques d'états, diminuant ainsi la complexité de l'algorithme. L'objectif en terme d'optimisation pourrait alors être de réexploiter ses techniques symboliques, en intégrant dans le processus de synthèse la notion de décision dynamique de reconfiguration (seule la décision statique est disponible pour le moment, via la *synthèse optimale de contrôleur*).

Optimisation de l'implémentation du contrôleur

L'exécution de la solution de contrôle de reconfiguration proposée est certes correcte, mais pourrait certainement gagner à être optimisée en étant implémentée matériellement. Cela permettrait notamment de s'affranchir du MicroBlaze, ici nécessaire pour mettre en œuvre physiquement les ordres de reconfiguration.

Nous supposons qu'une telle mise en œuvre occasionnerait deux types d'avantages :

– un éventuel gain de place, si l'implémentation matérielle du contrôleur nécessite moins de surface qu'un MicroBlaze ;
– une meilleure performance d'exécution ; si l'exécution d'un pas de contrôle est brève

(traverser son arbre des possibilités précalculé est peu complexe) comparée au reste des opérations effectuées dans un pas (notamment la mise en œuvre de la reconfiguration), une exécution complètement matérielle permettrait de repousser encore plus loin la cadence de contrôle : on pourrait imaginer un contexte d'exécution où les reconfigurations considérées prendraient peu de temps (exemple : simple changement de paramètres) et dans ce cas l'exécution d'un pas de contrôle serait critique à une fréquence élevée.

2.2 Perspectives d'intégration dans le projet FAMOUS

Proposition d'un démonstrateur réaliste

La perspective la plus évidente concerne l'objectif final du projet FAMOUS : la mise en œuvre d'un démonstrateur à la fois réaliste et capable d'illustrer l'ensemble des contributions. Nous contribuons à cette perspective de deux manières : 1) par la proposition d'une plateforme via laquelle il est simple de déployer des exemples d'applications reconfigurables tout à fait réalistes traitant un flux vidéo, et 2) par la démonstration de notre approche de contrôle sur cette plateforme.

L'objectif à terme du projet étant plus précisément de mettre en œuvre un exemple d'application de détection et suivi d'objets, la plateforme proposée va contribuer à illustrer plusieurs cas de dynamicité intéressants :

- la modification d'algorithmes de suivi d'objet ;
- la modification de la taille des zones de suivi sur le rendu vidéo ;
- l'adaptation du nombre de zones de suivi ;
- et bien sûr, la reconfiguration des *Processing Elements*, ce que nous avons illustré dans cette étude.

Pour l'instant, nous avons mis en œuvre le démonstrateur simple de notre étude sur cette plateforme, afin de valider la méthodologie défendue ici dans son ensemble. Il s'agit à présent pour le projet FAMOUS d'itérer sur cet exemple, afin d'intégrer progressivement les autres contributions.

Gestion de la diffusion des ordres de reconfiguration

Le module le plus couplé avec notre approche de contrôle est certainement la gestion des ordres de reconfiguration, que nous avons évoqué plusieurs fois sous le nom de *membrane*. Un mécanisme très simple de membrane à été mis en œuvre manuellement dans nos travaux, afin de configurer physiquement le système en conformité avec un ordre de reconfiguration reçu par un contrôleur. Il s'agissait ici de déclencher une configuration im-

pactant une tâche lorsque celle-ci termine son traitement dans une séquence donnée du modèle de calcul choisi (réseau de Kahn).

Il s'agit maintenant pour le module de la membrane d'itérer sur la solution proposée, en exploitant les routines présentes dans le framework de la plateforme afin de gérer :

- la diffusion d'un ordre de reconfiguration en suivant des contraintes imposées par un modèle de calcul ;
- la suspension et la reprise d'une tâche, si besoin ;
- la sauvegarde et la reprise de contexte, ce qui permettrait alors de présenter des cas de reconfiguration intéressants, avec gestion des cas où un ordre de reconfiguration échoue dans sa mise en place, obligeant la membrane à traiter la situation en suspendant (partiellement ou complètement) l'exécution du système afin de le réparer.

Contrôle de l'ordonnancement

Si notre proposition de contrôle est orthogonale à celle – proposée par un autre acteur du projet FAMOUS – concernant le contrôle optimal de l'ordonnancement, la plateforme que nous avons définie dans ces travaux doit pouvoir aider à illustrer ce type de contrôle.

Par défaut, nous disposons dans la partie logicielle de la plateforme d'une boucle d'exécution infinie, que nous avons d'ailleurs exploitée ici pour faire appel à notre contrôleur dans le démonstrateur. Cette boucle s'exécute en parallèle des tâches traitant le flux vidéo, ce qui illustre d'ailleurs l'indépendance contrôle/données. Or dans le cas du contrôle de l'ordonnancement, nous voudrions que le contrôle soit précisément couplé avec la cadence de traitement du flot de données, en effet, après chaque exécution d'une tâche, un pas de contrôle doit être appelé pour décider de la prochaine tâche à activer.

Plusieurs solution sont envisageables pour l'intégration de ce type de contrôle :

- la modification de la boucle principale d'exécution, afin de la rendre dépendante du flot de donnée en la synchronisant sur chaque fin de tâche ;
- l'intégration du contrôle en tant que PE, il est alors immédiatement synchronisé au flot de donnée puisqu'il fait implicitement partie des tâches traitant ce flot ;
- l'intégration du contrôle à la membrane : à noter que cette solution n'est pas à proprement parler une perspective concernant nos travaux, néanmoins l'idée d'exploiter les possibilités de la membrane – qui encapsule toute tâche du système afin de la piloter – peut être une piste intéressante à explorer.

Amélioration de la modélisation

Enfin, rappelons que d'une manière générale dans ce projet, c'est bien à l'obtention d'un flot complet de modélisation que nous aspirons ; un flot profitant au mieux des possibilités d'automatisation offertes par des transformations.

L'un des modules du projet FAMOUS est entièrement dédié à la modélisation dans MARTE, avec pour objectif de finaliser RecoMARTE. La responsabilité de ce module est de parvenir à intégrer la modélisation de tous les autres : la méthodologie IP-XACT pour l'intégration d'IP tierces, la membrane, le contrôle de l'ordonnancement, et notre module de contrôle de reconfiguration.

Dans cette étude, nous avons non seulement spécifié nos besoins en terme de contrôle, mais également fait une proposition de modélisation que nous avons concrétisée, avec extension du métamodèle MARTE, formalisation de transformations, et démonstration associée. Une telle proposition allège ainsi la charge du module de modélisation, et pourra être adaptée selon le cas afin d'assurer une cohérence avec le reste des contributions dans RecoMARTE (application, architecture, allocation).

La plateforme que nous avons définie dans ces travaux doit également servir de support à la modélisation : il s'agit d'un exemple concret et exécutable, potentiellement capable de démontrer l'ensemble des modules dans FAMOUS, et la perspective pour cette plateforme est d'être à terme complètement modélisée en RecoMARTE et IP-XACT, en vue de la générer automatiquement.

L'enjeu majeur de FAMOUS est à présent de mettre en œuvre le reste des moyens de modélisation et de déploiement de systèmes sur puce reconfigurables afin, nous l'espérons, de parvenir à combler le fossé de productivité dans la conception de ces systèmes.

Bibliographie

[1] Jason Agron. Domain-Specific Language for HW/SW Co-design for FPGAs. *DSL*, pages 262–284, 2009.

[2] Knut Akesson, Martin Fabian, and Hugo Flordal. Supremica in a Nutshell – Draft [online]. URL : `http://www.supremica.org/media/SupremicaNutshell.pdf`.

[3] Knut Akesson, Martin Fabian, Hugo Flordal, and Robi Malik. Supremica - An integrated environment for verification, synthesis and simulation of discrete event systems. *Workshop of Discrete Event Systems (WODES)*, pages 384–385, 2006.

[4] Karine Altisen, Aurélie Clodic, Florence Maraninchi, and Éric Rutten. *Using Controller-Synthesis Techniques to Build Property-Enforcing Layers*, volume 2618 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Berlin, Heidelberg, February 2003.

[5] Doug Amos, Austin Lesea, and René Richter. *FPGA-Based Prototyping Methodology Manual*. Best Practices in Design-For-Prototyping. Synopsys Press, March 2011.

[6] Xin An, Sarra Boumedien, Abdoulaye Gamatié, and Éric Rutten. CLASSY : a clock analysis system for rapid prototyping of embedded applications on MPSoCs. In *SCOPES '12 : Proceedings of the 15th International Workshop on Software and Compilers for Embedded Systems*. ACM Request Permissions, May 2012.

[7] Charles André. Representation and analysis of reactive behaviors : A synchronous approach. *Computational Engineering in Systems Applications (CESA)*, pages 19–29, 1996.

[8] Ralph-Johan Reinhold Back and Kaisa Sere. From action systems to modular systems. *FME'94 : Industrial Benefit of Formal Methods*, 1994.

[9] Kent Beck. *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, us ed edition, October 1999.

[10] Kent Beck. *Test Driven Development : By Example*. Addison-Wesley Professional, 1 edition, November 2002.

[11] Khaled Benkrid, Ali Akoglu, Cheng Ling, Yang Song, Ying Liu, and Xiang Tian. High Performance Biological Pairwise Sequence Alignment : FPGA versus GPU

versus Cell BE versus GPP. *International Journal of Reconfigurable Computing*, 2012.

[12] Jon Louis Bentley. *Programming Pearls*. Addison-Wesley Professional, 2000.

[13] Melanie Berg. Fault tolerance implementation within SRAM based FPGA designs based upon the increased level of single event upset susceptibility. *On-Line Testing Symposium*, 2006.

[14] Gérard Berry. The foundations of Esterel. *Proof, Language, and Interaction : Essays in Honour of Robin Milner*, pages 425–454, 2000.

[15] Brandon Blodget, Scott McMillan, and Patrick Lysaght. A lightweight approach for embedded reconfiguration of FPGAs. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 399–400. IEEE Computer Society, 2003.

[16] Pierre Bomel, Guy Gogniat, and Jean-Philippe Diguet. A networked, lightweight and partially reconfigurable platform. In *4th International Workshop, ARC*, London, UK, 2008.

[17] Tayeb Bouhadiba, Quentin Sabah, Gwenaël Delaval, and Éric Rutten. Synchronous control of reconfiguration in fractal component-based systems - A case study. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 309–318. ACM Request Permissions, 2011.

[18] Wray Lindsey Buntine and Bryan Preas. Design rule checking and analysis of IC mask designs. In *Proceedings of the 13th Design Automation Conference*, pages 301–308, New York, New York, USA, 1976. ACM Press.

[19] Paul Caspi, Grégoire Hamon, and Marc Pouzet. Lucid Synchrone, un langage de programmation des systèmes réactifs. In *Systèmes Temps-réel : Techniques de Description et de Vérification - Théorie et Outils*, pages 217–260. Hermes, 2006.

[20] Sana Cherif, Imran Rafiq Quadri, Samy Meftali, and Jean-Luc Dekeyser. Modeling Reconfigurable Systems-on-Chips with UML MARTE Profile : An Exploratory Analysis. *DSD*, pages 706–713, 2010.

[21] Pong P. Chu. *FPGA Prototyping by VHDL Examples.* Xilinx Spartan-3 Version. Wiley-Interscience, September 2011.

[22] Edmund Melson Clarke, Ernest Allen Emerson, and Aravinda Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, April 1986.

[23] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 1996.

[24] Philippe Coussy, Daniel Gajski, Michael Meredith, and Andres Takach. An Introduction to High-Level Synthesis. *Design & Test of Computers, IEEE*, 26(4) :8–17, 2009.

[25] Philippe Coussy and Adam Morawiec. *High-Level Synthesis.* From Algorithm to Digital Circuit. Springer Verlag, October 2008.

[26] John Curreri, Greg Stitt, and Alan George. High-level synthesis techniques for in-circuit assertion-based verification. *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, 2010.

[27] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional Transformations : A Cross-Discipline Perspective. *ICMT*, pages 260–283, 2009.

[28] Gwenaël Delaval. Répartition modulaire de programmes synchrones. Ph.-D. thesis, INRIA Rhône-Alpes, 2008.

[29] Gwenaël Delaval, Hervé Marchand, and Éric Rutten. Contracts for modular discrete controller synthesis. In *LCTES '10 : Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems.* ACM Request Permissions, April 2010.

[30] Ludovic Devaux, Sébastien Pillement, Daniel Chillet, and Daniel Demigny. Os services for reconfigurable system-on-chip communications. *Design of Circuits and Integrated Systems*, 2010.

[31] Jean-Philippe Diguet, Yvan Eustache, and Guy Gogniat. Closed-loop based self-adaptive HW/SW embedded systems : design methodology and smart cam case study. *ACM Trans. Embed. Comput. Syst.*, 2010.

[32] Apostolos Dollas, Brent Ward, and John Daniel Sterling Babcock. *Lecture Notes in Computer Science*, volume 849 of *GerhardGoosJurisHartmanisLecture Notes in Computer Science0302-97431611-3349.* Springer Berlin Heidelberg, Berlin, Heidelberg, June 2005.

[33] Micah Dowty. Test driven development of embedded systems using existing software test infrastructure. *University of Colorado at Boulder*, 2004.

[34] Emil Dumitrescu, Alain Girault, Hervé Marchand, and Éric Rutten. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. In *Workshop on Discrete Event Systems, WODES'10*, pages 366–373, Berlin, Allemagne, 2010. Ampère , POP ART - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble , VERTECS - INRIA , SARDES - INRIA Grenoble Rhône-Alpes / LIG Laboratoire d'Informatique de Grenoble, IFAC.

[35] Carl Ebeling, Chris Fisher, Guanbin Xing, Manyuan Shen, and Hui Liu. Implementing an OFDM receiver on the RaPiD reconfigurable architecture. *Computers, IEEE Transactions on*, 53(11) :1436–1448, 2004.

[36] Sven Eisenhardt, Tobias Oppold, Thomas Schweizer, and Wolfgang Rosenstiel. Optimizing Partial Reconfiguration of Multi-context Architectures. In *Reconfigurable Computing and FPGAs, 2008. ReConFig '08. International Conference on*, pages 67–72, 2008.

[37] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The Promise of High-Performance Reconfigurable Computing. *Computer*, 41(2) :69–76, 2008.

[38] Gerald Estrin. Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference on - IRE-AIEE-ACM '60 (Western). In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, page 33, New York, New York, USA, 1960. ACM Press.

[39] Madeleine Faugère, Thimothée Bourbeau, Robert de Simone, and Sébastien Gérard. MARTE : Also an UML Profile for Modeling AADL Applications. In *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 359–364, 2007.

[40] Sahar Foroutan, Yvain Thonnart, Richard Hersemeule, and Ahmed Jerraya. A Markov chain based method for NoC end-to-end latency evaluation. In *Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[41] Abdoulaye Gamatié, Sébastien Le Beux, Éric Piel, Rabie Ben Atitallah, Anne Etien, Philippe Marquet, and Jean-Luc Dekeyser. A model-driven design framework for massively parallel embedded systems. *ACM Trans. Embed. Comput. Syst.*, 10(4) :39 :1–39 :36, November 2011. URL : `http://doi.acm.org/10.1145/2043662.2043663`, `doi:10.1145/2043662.2043663`.

[42] Erich Gamma. *Design Patterns.* Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1995.

[43] Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Syst. Signal Process.*, 2010.

[44] Guy Gogniat, Jorgiano Vidal, Linfeng Ye, Jérémie Crenne, Sébastien Guillet, Florent de lamotte, Jean-Philippe Diguet, and Pierre Bomel. Self-reconfigurable embedded systems : from modeling to implementation. *Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2010.

[45] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R Reed Taylor, and Ronald Laufer. PipeRench : A Coprocessor for Streaming multimedia Acceleration. *ISCA*, pages 28–39, 1999.

[46] Mentor Graphics. ModelSim [online]. URL : `http://model.com/`.

[47] Object Management Group. Meta Object Facility [online]. URL : `http://www.omg.org/mof/`.

[48] Object Management Group. Model Driven Architecture [online]. URL : `http://www.omg.org/mda/`.

[49] Object Management Group. Modeling and analysis of real-time and embedded systems (MARTE) [online]. URL : `http://www.omgmarte.org`.

[50] Object Management Group. Unified Modeling Language specification [online]. URL : `http://www.omg.org/spec/UML/`.

[51] Object Management Group. UML profile for Schedulability, Performance and Time (SPT). *Version 1.1*, 2005.

[52] Object Management Group. UML profile for System on Chip (SoC). *Version 1.0.1*, 2006.

[53] GTKWave. Official website [online]. URL : `http://gtkwave.sourceforge.net/`.

[54] Sébastien Guillet, Florent de lamotte, Nicolas Le Griguer, Éric Rutten, Jean-Philippe Diguet, and Guy Gogniat. Designing formal reconfiguration control using UML/MARTE. *7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, 2012.

[55] Sébastien Guillet, Florent de lamotte, Nicolas Le Griguer, Éric Rutten, Jean-Philippe Diguet, and Guy Gogniat. Modeling and synthesis of a Dynamic and Partial Reconfiguration controller. *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012.

[56] Sébastien Guillet, Florent de lamotte, Éric Rutten, Guy Gogniat, and Jean-Philippe Diguet. Modeling and Formal Control of Partial Dynamic Reconfiguration. *Reconfigurable Computing and FPGAs (ReConFig)*, 2010.

[57] Sébastien Guillet, Florent de lamotte, Éric Rutten, Guy Gogniat, and Jean-Philippe Diguet. Modélisation et contrôle de la reconfiguration dynamique et partielle. *SYMPosium en Architectures (SYMPA)*, 2011.

[58] Sébastien Guillet and Loïc Lagadec. Ajout d'un système de types à un atelier de synthèse de circuits. *MajecSTIC*, 2008.

[59] Nicholas Halbwachs. Synchronous programming of reactive systems. Kluwer Academic, 1993.

[60] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. In *Proceedings of the IEEE*, 1991.

[61] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 1996.

[62] David Harel and Amir Pnueli. On the development of reactive systems. Logic and Models of Concurrent Systems, NATO, 1985.

[63] Reiner Hartenstein. A decade of reconfigurable computing : a visionary retrospective. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 642–649, 2001.

[64] Scott Hauck. The future of reconfigurable systems. *5th Canadian Conference on Field Programmable Devices*, 1998.

[65] Gerard Hoffmann and Howard Wong-Toi. Symbolic synthesis of supervisory controllers. *American Control Conference*, 1992.

[66] William Albert Howes. On-orbit FPGA SEU Mitigation and Measurement Experiments on the Cibola Flight Experiment Satellite. Master Thesis, Brigham Young University, 2011.

[67] IABG. V-Modell [online]. URL : `http://v-modell.iabg.de/`.

[68] Mohamed Mahmoud Ibrahim, Ahmed Mahmoud Tobal, Mohamed Youssri El Nahas, and Mohamed Refai. FPGA based on board computer for LEO satellites. In *Space Science and Communication (IconSpace), 2011 IEEE International Conference on*, pages 314–319, 2011.

[69] Cadence Design Systems Inc. Cadence System Development Suite [online]. URL : `http://www.cadence.com`.

[70] INRIA. CADP [online]. URL : `http://cadp.inria.fr/`.

[71] IRISA. Sigali [online]. URL : `http://www.irisa.fr/vertecs/Logiciels/sigali.html`.

[72] Vatche Ishakian and Azer Bestavros. MORPHOSYS : Efficient Colocation of QoS-Constrained Workloads in the Cloud. *CCGRID*, pages 90–97, 2012.

[73] Stephen Johnson. Lint, a C program checker. *Computer science technical report*, 1978.

[74] Gilles Kahn. *The semantics of a simple language for parallel programming.* Proceedings of IFIP Congress, 1974.

[75] Stuart Kent. *Model Driven Engineering*, volume 2335 of *IFM '02*. Proceedings of the Third International Conference on Integrated Formal Methods, Berlin, Heidelberg, April 2002.

[76] Mickael Kerboeuf, Alain Plantec, and Jean-Philippe Babau. An experiment of a MDE approach for the design of reusable DSL tools. *Actes des journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, pages 25–30, June 2011.

[77] Markus Koester, Wayne Luk, Jens Hagemeyer, and Mario Porrmann. Design optimizations to improve placeability of partial reconfiguration modules. In *Design, Automation & Test in Europe Conference & Exhibition, DATE'09*, pages 976–981, 2009.

[78] Wolfgang Kranz. An Integrated System Development Process including Hardware and Logistics based on a Standard Software Process Model. *Technology for Evolutionary Software Development*, 2003.

[79] Eren Kursun and Chen-Yong Cher. Variation-aware thermal characterization and management of multi-core architectures. In *IEEE International Conference on Computer Design (ICCD)*, pages 280–285, 2008.

[80] Philippe Lacan, Jean Noël Monfort, Le Vin Quy Ribal, Alain Deutsch, and Georges Gonthier. ARIANE 5 - The Software Reliability Verification Process. *Data Systems in Aerospace (DASIA)*, 1998.

[81] Loïc Lagadec, Bernard Pottier, and Oscar Villellas-Guillen. *An LUT-based high level synthesis framework for reconfigurable architectures*. Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation, 2003.

[82] Sébastien Le Beux, Philippe Marquet, and Jean-Luc Dekeyser. Model Driven Engineering Benefits for High Level Synthesis. *Rapport de recherche INRIA*, 2008.

[83] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming Real Time Applications with SIGNAL. Proceedings of the IEEE, 1991.

[84] Gerard Le Lann. The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems. *INRIA research report*, 1996.

[85] Stéphane Lecomte, Samuel Guillouard, Christophe Moy, Pierre Leray, and Philippe Soulard. A co-design methodology based on model driven architecture for real time embedded systems. *Mathematical and Computer Modelling : An International Journal*, 53(3-4), February 2011.

[86] Éric Lemoine and David Merceron. Run time reconfiguration of FPGA for scanning genomic databases. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 90–98, 1995.

[87] Stefan Leue and Wei Wei. Integer Linear Programming Based Property Checking for Asynchronous Reactive System. *IEEE Transactions on Software Engineering*, 2011.

[88] Nancy Leveson and Clark Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7) :18–41, 1993.

[89] Felix Madlener, Alexander Biedermann, and Sorin Huss. RecDEVS : A Comprehensive Model of Computation for Dynamically Reconfigurable Hardware Systems. *4th IFAC Workshop on Discrete-Event System Design*, 2009.

[90] Florence Maraninchi and Yann Rémond. Mode-automata : About modes and states for reactive systems. *European Symposium On Programming*, 1998.

[91] Florence Maraninchi and Yann Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, 27(1-3), April 2001.

[92] Florence Maraninchi and Yann Rémond. Mode-Automata : a new domain-specific construct for the development of safe critical systems. *Science of computer programming*, 46(3) :219–254, March 2003.

[93] Pierre-Alain Masson. Vérification par model-checking modulaire de propriétés dynamiques PLTL exprimées dans le cadre de spécifications B événementielles. *Ph.-D. Thesis, Université de Franche-Comté*, 2001.

[94] John McDermid and Knut Ripken. Life cycle support in the ADA environment. ACM SIGAda Ada Letters, 1983.

[95] Kenneth McMillan. *Symbolic Model Checking, an approach to the state explosion problem.* Ph.-D. thesis, Carnegie Mellon University, May 1992.

[96] Carver Mead and Lynn Conway. Introduction to VLSI Systems. Addison-Wesley Longman Publishing Co., 1980.

[97] George Mealy. *A method for synthesizing sequential circuits.* Bell System Technical Journal, 1955.

[98] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4) :316–344, December 2005.

[99] Matthias Merten. *Design of Interactive Service Robots applying methods of Systems Engineering and Decision Making.* Ph.-D. thesis, Technischen Universität Ilmenau, 2012.

[100] Joaquin Miller and Jishnu Mukerji. Model driven architecture. *Tech. rep., Object Management Group*, 2003.

[101] Fabrice Muller, Jimmy Le Rhun, Fabrice Lemonnier, Benoît Miramond, and Ludovic Devaux. A Flexible Operating System for Dynamic Applications. *Xcell Journal, n° 73*, 2010.

[102] Marcello Mura, Luis Gabriel Murillo, and Mauro Prevostini. Model-based Design Space Exploration for RTES with SysML and MARTE. *Specification, Verification and Design Languages, 2008. FDL 2008. Forum on*, pages 203–208, 2008.

[103] Surya Narayanan, Daniel Chillet, Sébastien Pillement, and Ioannis Sourdis. Hardware OS Communication Service and Dynamic Memory Management for RSoCs. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 117–122, 2011.

[104] Hùng Nguyen. Rapid prototyping using Field Programmable Gate Array (FPGA) and Field Programmable Interconnect Devices (FPID), 1996.

[105] Gilberto Ochoa, El-Bay Bourennane, Ouassila Labbani, and Kamel Messaoudi. IP-XACT and marte based approach for partially reconfigurable systems-on-chip. *FDL*, pages 1–8, 2011.

[106] Peter Øhrstrøm and Peter Hasle. *Temporal Logic : From Ancient Ideas to Artificial Intelligence*. Studies in Linguistics and Philosophy. Springer, September 1995.

[107] Oliver Pell. Verification of FPGA Layout Generators in Higher-Order Logic. *Journal of Automated Reasoning*, 2006.

[108] Sébastien Pillement. Conception d'architectures reconfigurables dynamiquement : Du silicium au système. *Habilitation thesis, Université de Rennes 1*, 2010.

[109] Christian Plessl and Marco Platzner. Virtualization of hardware–introduction and survey. *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04)*, 2004.

[110] Johan Pouwelse, Koen Langendoen, and Henk Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th annual international conference on Mobile computing and networking*, 2001.

[111] Imran Rafiq Quadri. MARTE Based Model Driven Design Methodology for Targeting Dynamically Reconfigurable FPGA Based SoCs. Ph.-D. thesis, Université des Sciences et Technologie de Lille, 2010.

[112] Imran Rafiq Quadri, Abdoulaye Gamatié, Pierre Boulet, Samy Meftali, and Jean-Luc Dekeyser. Expressing embedded systems configurations at high abstraction levels with UML MARTE profile : Advantages, limitations and alternatives. *Journal of Systems Architecture : the EUROMICRO Journal*, 58(5), April 2012.

[113] Jean-Pierre Queille and Joseph Sifakis. A temporal logic to deal with fairness in transition systems. *23rd Annual Symposium on Foundations of Computer Science, SFCS '08*, pages 217–225, 1982.

[114] P. J.G. Ramadge. *Modular feedback logic for discrete event systems.* SIAM Journal on Control and Optimization, 1987.

[115] P. J.G. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *Analysis and Optimization of Systems*, 1984.

[116] Rajeshuni Ramesham, Justin N Maki, Ali M Pourangi, and Steven W Lee. Qualification of Engineering Camera for Long-Duration Deep Space Missions, 2012.

[117] David Ratter. FPGAs on mars. *Xcell J*, 2004.

[118] Pascal Raymond. Lustre manual V4 [online]. URL : `http://www-verimag.imag.fr/~raymond/tools/lv4_man.ps.gz`.

[119] Elvinia Riccobene, Patrizia Scandurra, Alberto Rosti, and Sara Bocchio. A model-driven design environment for embedded systems. *DAC*, pages 915–918, 2006.

[120] Laurent Rioux and Madeleine Faugère. MARTE & AADL : Mise en correspondance des concepts. *Génie Logiciel n° 97*, pages 4–8, 2011.

[121] Winston Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, 1970.

[122] Éric Rutten and Paul Le Guernic. Sequencing date flow tasks in SIGNAL. *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[123] Éric Rutten, Éric Marchand, and François Chaumette. An experiment with reactive data-flow tasking in active robot vision. *Software—Practice & Experience*, 27(5), May 1997.

[124] Thomas Santen and Dirk Seifert. Executing UML state machines. Technical Report 2006-04, Fakultät für Elektrotechnik und Informatik, Technische Universität Berlin, 2006.

[125] Ulrich Schiel and Ivan Mistrik. Using object-oriented analysis and design for integrated systems. In *Systems Integration, 1990. Systems Integration '90., Proceedings of the First International Conference on*, pages 125–134, 1990.

[126] Douglas Schmidt. Guest Editor's Introduction : Model-Driven Engineering. *Computer*, 39(2) :25–31, 2006.

[127] SDMetrics. Software Design Metrics tool for the UML [online]. URL : `http://www.sdmetrics.com/`.

[128] Marco Sgroi, Michael Sheets, Andrew Mihal, Kurt Keutzer, Sharad Malik, Jan Rabaey, and Alberto Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. In *Design Automation Conference, 2001. Proceedings*, pages 667–672. ACM Request Permissions, 2001.

[129] Sim2Chro. Official website [online]. URL : `http://www.di.ens.fr/~pouzet/lucid-synchrone/manual_html/manual038.html`.

[130] Efstathios Sotiriou-Xanthopoulos, Ioannis Koutras, George Economakos, and Dimitrios Soudris. A reconfigurable IP characterization technique improving high-level synthesis results. In *6th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–6, 2011.

[131] B. Sreekandath and B. Priyadarshan. An integrated flow for ASIC designs with FPGA prototyping-a designer's perspective. In *WESCON/'95. Conference record. 'Microelectronics Communications Technology Producing Quality Products Mobile and Portable Power Emerging Technologies*, 1995.

[132] SystemC. The Open SystemC Initiative [online]. URL : `http://www.systemc.org`.

[133] Florian Thoma, Matthias Kuhnle, Philippe Bonnot, Elena Moscu Panainte, Koen Bertels, Sebastian Goller, Axel Schneider, Stephane Guyetant, Eberhard Schuler, Klaus D Muller-Glaser, and Jürgen Becker. MORPHEUS : Heterogeneous Reconfigurable Computing. In *International Conference on Field Programmable Logic and Applications*, pages 409–414. IEEE, 2007.

[134] Alan Mathison Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction. *Proceedings of the London Mathematical Society*, s2-43(6) :544–546, January 1938.

[135] Raoul Velazco, Pascal Fouillat, and Ricardo Augusto da Luz Reis. *Radiation Effects on Embedded Systems*. Springer Verlag, May 2007.

[136] Jorgiano Vidal. *Dynamic and partial reconfigurable embedded systems design with uml*. Ph.-D. Thesis, Université de Bretagne Sud, 2010.

[137] Jorgiano Vidal, Florent de lamotte, Guy Gogniat, Jean-Philippe Diguet, and Sébastien Guillet. Dynamic applications on reconfigurable systems : From UML model design to FPGAs implementation. *Design, Automation and Test in Europe (DATE)*, 2011.

[138] Pamela Wattebled, Jean-Philippe Diguet, and Jean-Luc Dekeyser. Membrane-based design and management methodology for parallel dynamically reconfigurable embedded systems. In *ReCoSoC*, 2012.

[139] Makarius Wenzel. The Isabelle/Isar Reference Manual [online]. 2012. URL : `http://isabelle.in.tum.de/doc/isar-ref.pdf`.

[140] Dongmok Whang, Song Jin, and Charles M Lieber. Nanolithography Using Hierarchically Assembled Nanowire Masks. *Nano Letters*, 3(7) :951–954, July 2003.

[141] Xilinx Inc. Early Access Partial Reconfigurable Flow [online]. URL : `http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm`.

[142] Xilinx Inc. Xilinx Virtex 7 family [online]. URL : `http://www.xilinx.com/products/silicon-devices/fpga/virtex-7/index.htm`.

[143] Linfeng Ye, Jean-Philippe Diguet, and Guy Gogniat. Reconfigurable MPSoCs for On-Demand Computing. *GRETSI*, 2009.

[144] Huafeng Yu. A MARTE based reactive model for data-parallel intensive processing : Transformation toward the synchronous model. Ph.-D. Thesis, Université des Sciences et Technologie de Lille, 2008.

[145] Huafeng Yu, Abdoulaye Gamatié, and Éric Rutten. Safe design of high-performance embedded systems in an MDE framework. *Innovations in Systems and Software Engineering*, 2008.

[146] Yijun Yu, Yu Lin, Zhenjiang Hu, Soichiro Hidaka, Hiroyuki Kato, and Lionel Montrieux. Maintaining invariant traceability through bidirectional transformations. *ICSE*, pages 540–550, 2012.

[147] Janan Zaytoon. *Systèmes Dynamiques Hybrides*. Hermes, 2001.

[148] Zhonghua Zhang and W. M. Wonham. STCT : An efficient algorithm for supervisory control design. *Symposium on Supervisory Control of Discrete Event Systems*, 2001.

Résumé

Cette thèse a pour objet l'étude de la modélisation du contrôle de la reconfiguration dans les systèmes dynamiques, plus particulièrement les systèmes sur puce dynamiquement et partiellement reconfigurables. Les travaux présentés dans ce manuscrit visent à réaliser une méthodologie de conception *par contrainte* du contrôle, applicable dans le cadre de la spécification de ces systèmes. Reposant sur le principe d'Ingénierie Dirigée par les Modèles, cette méthodologie – basée sur UML/MARTE – est dotée de transformations appropriées, lui permettant de cibler une représentation synchrone, en langage BZR, de la partie contrôle. Cette représentation est ensuite exploitable par une technique correcte par construction – la synthèse de contrôleur discret –, dans le but d'obtenir automatiquement et de manière sûre les lois de commande correspondant aux contraintes spécifiées en amont. La partie contrôle est plus particulièrement divisée en deux aspects : la *sécurité*, obtenue formellement par synthèse afin de produire des espaces de configurations accessibles, et l'*optimisation*, implémentable par le concepteur et produisant un ordre de reconfiguration à partir d'un espace accessible. L'intégration sécurité/optimisation proposée est assimilable à un système réactif avec boucle de rétroaction. Un exemple démontrant la méthodologie est réalisé, et fait apparaître ses avantages tant en terme de simplification de conception (spécification par contraintes, approche automatique) qu'en terme de sécurité (contrôle formel).

Mots-clés: Ingénierie Dirigée par les Modèles, UML, MARTE, système réactif, boucle de rétroaction, langage synchrone, BZR, méthode formelle, Synthèse de Contrôleur Discret.

Abstract

This thesis deals with the study of reconfiguration control modeling in dynamic systems, especially dynamically and partially reconfigurable Systems-on-Chip. The work presented in this manuscript aims to carry out a *design-by-constraints* methodology, applicable to the specification of these systems. Relying on Model Driven Engineering, this methodology – based on UML/MARTE – is provided with appropriated transformations, enabling it to target a synchronous representation, in the BZR language, dedicated to the control part. This representation is in turn operated through a correct-by-construction technique – discrete controller synthesis –, which aims to automatically and safely obtain control laws corresponding to the previously specified constraints. The control part is divided into two aspects : *safety*, formally obtained through synthesis and providing accessible configuration space, and *optimisation*, implementable by the designer and providing a reconfiguration order considering an accessible configuration space. The proposed incorporation of safety and optimisation is comparable to a reactive system using a feedback loop. An example showing the methodology is presented, illustrating its benefits on both the design simplification (designing by constraints, automated approach) and security levels (formal control).

Keywords: Model Driven Engineering, UML, MARTE, reactive system, feedback loop, synchronous language, BZR, formal method, Discrete Controller Synthesis.