THÈSE de DOCTORAT de l'UNIVERSITÉ PARIS 6

Spécialité

INFORMATIQUE

présentée par

Nicolas Pernet

pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PARIS 6

Sujet de la thèse :

Implantation distribuée temps réel de programmes conditionnés à l'aide d'ordonnancements mixtes hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques

soutenue le 7 juillet 2006

devant le jury composé de

Pascale Minet	Chargée de Recherche à l'INRIA, HDR	Présidente du jury
Fabrice Kordon	Professeur à l'Université Paris 6	Directeur de thèse
Yves Sorel	Directeur de Recherche à l'INRIA	Encadrant
Bertil Folliot	Professeur à l'Université Paris 6	Examinateur
Maryline Silly-Chetto	Professeur à l'IUT de Nantes	Rapporteur
Elie Najm	Professeur à l'ENST	Rapporteur

Implantation distribuée temps réel de programmes conditionnés à l'aide d'ordonnancements mixtes hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques

Nicolas Pernet

Version du 3 juillet 2006

Remerciements

Je remercie en premier lieu Fabrice Kordon, mon directeur de thèse, pour la confiance qu'il m'a témoigné. Tout en déléguant une partie du suivi de mon doctorat à Yves Sorel, mon encadrant INRIA, il a su apporter un regard critique sur mes travaux et se montrer disponible pour m'écouter et me conseiller.

Tout naturellement, je remercie Yves Sorel de m'avoir accueilli à l'INRIA d'abord en stage de DEA, puis de m'avoir permis de poursuivre les travaux amorcés durant ces six mois par un doctorat. Encadrant idéal, il a su guider mes travaux tout en me laissant une grande autonomie. J'ai notamment acquis auprès de lui la rigueur scientifique qui faisait défaut dans mes présentations et rédactions.

Je tiens à remercier également Maryline Silly-Chetto et Elie Najm qui ont eu la gentillesse de bien vouloir être les rapporteurs de cette thèse. De même, j'adresse un grand merci à Pascal Minet et Bertil Folliot pour avoir accepté de faire partie de mon jury de thèse.

Ayant au départ suivi une formation d'ingénieur, vous ne liriez pas ces lignes si de longues discussions avec Jean-Marie Gilliot et Rémy Kocik enseignants-chercheurs à l'École Supérieure d'Ingénieurs en Électronique et Électrotechnique ne m'avaient pas donné envie de devenir enseignant-chercheur. Je remercie tout particulièrement Rémy qui m'a présenté à Yves Sorel et m'a proposé, en accord avec ce dernier, un stage sur mesure.

Durant ces quatre années de thèse effectuée au sein de l'action OSTRE devenue projet AOSTE, j'ai cotoyé doctorants, ingénieurs et stagiaires, les uns et les autres se succédant, changeant parfois de statut au fil des ans. Je remercie Liliana pour m'avoir guidé dans mes premiers pas de doctorant alors qu'elle finissait son doctorat. Un grand merci à Julien, mon collègue de bureau pendant de trop courtes années, ainsi que Cyril, Arnaud, Olivier, Christophe, Xavier et Benoît. Plus que des collègues, ils sont devenus des amis.

Je remercie également les membres actuels de l'équipe. Ils m'ont toujours soutenu même si je n'étais pas toujours disponible pour eux durant cette dernière année de thèse.

Un grand meci enfin à Karine et ma famille qui m'ont supporté, dans tous les sens du terme, durant ce parcours initiatique qu'est le doctorat.

Table des matières

Remerciements **3**

Introduction **15**

I Implantation distribuée de programmes conditionnés **21**

Introduction de la partie I **23**

1 État de l'art **25**
1.1 Concepts de base 25
1.2 Deux approches différentes pour décrire un algorithme 27
1.3 Modèle flot de contrôle 27
1.4 Modèle flot de données 28
1.5 Différences entre les deux modèles 30
1.6 Utilisation conjointe des deux modèles 32
1.7 Implantation distribuée de programmes flot de contrôle et flot de données 33
1.7.1 Implantation distribuée de programmes séparés 33
1.7.2 Unification de programmes 33
1.7.3 Implantation distribuée d'un programme flot de données 35
1.8 Conclusion de l'état de l'art 35

2 Modèle flot de données permettant le conditionnement pour implantation distribuée **37**
2.1 Limites des modèles flot de données existants permettant le conditionnement . . . 37
2.2 Modèle flot de données conditionné 39
2.2.1 Opération conditionnante 40
2.2.2 Dépendance conditionnée et de conditionnement 41
2.2.3 Ajout de dépendances et d'opérations pour implantation distribuée 43
2.2.3.1 D'opération conditionnante à opération conditionnée 43
2.2.3.2 Cas des dépendances conditionnées entrantes 44
2.2.3.3 Cas des dépendances conditionnées internes 48
2.2.3.4 Cas des dépendances conditionnées sortantes 48
2.2.3.5 Opérations conditionnantes conditionnées 49

2.3 SynDEx, langage utilisant le modèle flot de donnée conditionné 52

3 Traductions de langages permettant le conditionnement en langage SynDEx 53
3.1 Nécessité de traductions 53
3.2 Traduction SyncCharts/SynDEx 53
3.2.1 Des FSM non hiérarchiques à SyncCharts 53
3.2.2 Le langage SyncCharts 55
3.2.3 Principes de la traduction 57
3.2.3.1 Traduction d'une constellation 58
3.2.3.2 Traduction de la composition parallèle 63
3.2.3.3 Traduction d'un macro-état (hiérarchie) 63
3.2.3.4 Traduction des signaux locaux 66
3.2.3.5 Traduction de la préemption faible 67
3.2.3.6 Traduction de la terminaison normale 69
3.2.3.7 Traduction de la suspension 72
3.2.3.8 Traduction de l'arc instantané 72
3.2.3.9 Signaux valués et opérations 74
3.2.3.10 Conclusion 75
3.3 Traduction Scicos/SynDEx 76
3.3.1 Le langage Scicos 76
3.3.1.1 Les signaux 77
3.3.1.2 Rémanence de donnée dans Scicos 78
3.3.1.3 Relations entre activations 80
3.3.2 Fonctionnalités additionnelles 82
3.3.2.1 Blocs sources et blocs puits dans une schéma-bloc Scicos 82
3.3.2.2 "Region to superbloc" 82
3.3.2.3 La pré-compilation des schémas-blocs Scicos 83
3.3.3 Principes de la traduction 84
3.3.3.1 Divergences principales entre les langages 84
3.3.3.2 Restrictions sur les schémas-blocs pouvant être traduits 85
3.3.3.3 Principes de base 86
3.3.3.4 Cas simple 88
3.3.3.5 Caractérisation des relations entre signaux d'activation 89
3.3.3.6 Hypergraphe acyclique orienté d'activation 91
3.3.3.7 Exemple 100
3.3.3.8 Conclusion 100

4 Implantation distribuée avec le logiciel SynDEx 103
4.1 Principes du logiciel SynDEx 103
4.2 Le modèle flot de données conditionné dans le logiciel SynDEx 107
4.2.1 Description du conditionnement avec le langage SynDEx 107
4.2.2 Mise à plat du graphe SynDEx avant distribution 108
4.2.3 Distribution du conditionnement avec les heuristiques 109

4.3 Avantages de SynDEx vis à vis des logiciels existants . . . 110

5 Exemples d'implantations distribuées de programmes conditionnés 115
5.1 Traducteur Scicos/SynDEx . . . 115
5.2 Exemple d'implantation distribuée d'algorithme décrit avec Scicos . . . 117
5.3 Exemple d'implantation distribuée d'algorithme décrit conjointement par SyncCharts et Scicos . . . 118
5.3.1 Description du système . . . 118
5.3.2 Description de l'algorithme avec SyncCharts et Scicos . . . 118
5.3.3 Traduction et unification des spécifications . . . 119
5.3.4 Implantation distribuée avec SynDEx . . . 119

Conclusion de la partie I 129

II Ordonnancement temps réel mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques 131

Introduction de la partie II 133

6 État de l'art 135
6.1 Tâches périodiques, apériodiques et sporadiques . . . 135
6.1.1 Définitions . . . 135
6.1.2 Modèle temps réel classique et notations . . . 138
6.1.2.1 Tâche périodique . . . 138
6.1.2.2 Tâche apériodique . . . 139
6.1.2.3 Contraintes de précédence . . . 139
6.1.2.4 Ordonnancement et ordonnançabilité . . . 140
6.1.3 Ordonnancement de tâches périodiques . . . 140
6.1.3.1 Algorithmes à priorités fixes . . . 141
6.1.3.2 Algorithmes à priorités dynamiques . . . 143
6.1.3.3 Choix d'un algorithme . . . 144
6.1.4 Ordonnancement de tâches apériodiques . . . 144
6.1.5 Ordonnancement de tâches sporadiques . . . 146
6.2 Hors-ligne et en-ligne . . . 147
6.3 Ordonnancement mixte de tâches périodiques et apériodiques . . . 149
6.3.1 Approches à priorités fixes . . . 151
6.3.1.1 Serveur à scrutation . . . 151
6.3.1.2 Serveur ajournable . . . 152
6.3.1.3 Serveur à échange de priorités . . . 154
6.3.1.4 Serveur sporadique . . . 155
6.3.1.5 Slack stealing . . . 157
6.3.2 Approches à priorités dynamiques . . . 158

6.3.2.1 Serveur à scrutation dynamique . . . 158
6.3.2.2 Serveur ajournable dynamique . . . 158
6.3.2.3 Serveur à échange de priorités dynamique . . . 159
6.3.2.4 Serveur sporadique dynamique . . . 160
6.3.2.5 Serveur à utilisation totale . . . 162
6.3.2.6 Serveur Earliest Deadline Late . . . 163
6.3.2.7 Serveur à échange de priorités dynamiques amélioré . . . 165
6.3.2.8 Versions optimales et améliorées du serveur à utilisation totale . 165
6.3.3 Approches indépendantes de l'algorithme d'ordonnancement . . . 166
6.3.3.1 Slot shifting . . . 166
6.3.3.2 Feedback scheduling . . . 167
6.3.4 Cas distribué . . . 167
6.3.5 Optimalité et performances . . . 168
6.3.5.1 Optimalité . . . 168
6.3.5.2 Performances des approches à priorités fixes . . . 171
6.3.5.3 Performances des approches à priorités dynamiques . . . 174
6.3.5.4 Comparatif des différentes approches . . . 174

7 Contraintes de latence et ordonnancement mixte hors-ligne en-ligne 177
7.1 Limites du modèle temps réel classique . . . 177
7.2 Définition de la contraintes de latence . . . 180
7.3 Problème à résoudre . . . 181
7.4 Réduction de l'ensemble des contraintes à respecter . . . 182
7.4.1 Principes . . . 182
7.4.2 Uniformisation des échéances . . . 183
7.4.3 Contrainte de latence respectée si une échéance est respectée . . . 183
7.4.4 Contrainte de latence respectée si une autre contrainte de latence est respectée 184
7.4.5 Contrainte de latence respectée si d'autres contraintes de latence sont respectées . . . 185
7.4.6 Exemple . . . 186

8 Ordonnancement temps réel mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques 189
8.1 Choix de la méthode . . . 189
8.2 Principes du slot shifting . . . 190
8.2.1 Modèles . . . 190
8.2.2 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques 191
8.2.2.1 Dates de début au plus tôt et échéances . . . 191
8.2.2.2 Intervalles d'exécution et spare capacities . . . 191
8.2.3 Calculs en-ligne pour l'acceptation de tâches apériodiques . . . 193
8.2.3.1 Prise en compte des tâches apériodiques . . . 193
8.2.3.2 Test d'acceptation . . . 194
8.2.3.3 Ordonnancement en-ligne . . . 196

8.2.3.4 Mises à jour après ordonnancement en-ligne . . . 196
8.2.3.5 Exemple . . . 197
8.2.4 Optimalité . . . 200
8.3 Contraintes de latence et slot shifting . . . 201
8.4 Slot shifting où les latences sont traduites hors-ligne par des échéances mobiles . . 204
8.4.1 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques avec contraintes de latence . . . 204
8.4.2 Calculs en-ligne pour l'acceptation de tâches apériodiques . . . 205
8.4.2.1 Test d'acceptation . . . 205
8.4.2.2 Ordonnancement en-ligne . . . 208
8.4.3 Complexité . . . 209
8.4.4 Optimalité . . . 209
8.5 Slot shifting où les latences sont traduites en ligne par des échéances fixes . . . 210
8.5.1 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques avec contraintes de latence . . . 210
8.5.1.1 Cas $d_i \neq d_j$ . . . 210
8.5.1.2 Cas $d_i = d_j$ . . . 212
8.5.1.3 Construction de $\mathcal{L}_{hl}$ . . . 216
8.5.2 Calculs en-ligne pour l'acceptation de tâches apériodiques . . . 217
8.5.2.1 Acceptation de tâches apériodiques fermes et exécution de tâches apériodiques souples . . . 218
8.5.2.2 Exécution de tâches apériodiques souples seulement . . . 219
8.5.3 Complexité . . . 220
8.5.3.1 Acceptation de tâches apériodiques fermes et exécution de tâches apériodiques souples . . . 220
8.5.3.2 Exécution de tâches apériodiques souples seulement . . . 220
8.5.4 Optimalité . . . 221

Conclusion de la partie II **223**

Conclusion générale et perspectives **225**

Table des figures

1 Objectifs de la thèse relativement à l'implantation distribuée des systèmes temps réel 19

1.1 Exemple de description de parallélisme avec des FSM 31

2.1 Exemple de programme Integer Data Flow 38
2.2 Graphe flot de données conditionné 40
2.3 Exemple de distribution et d'ordonnancement du graphe de la figure 2.2 page 40 sur une architecture distribuée 42
2.4 Transformation des dépendances entrantes $E \rightarrow C_{11}$ et $E \rightarrow C_{21}$ de la figure 2.2 page 40 48
2.5 Transformation de la dépendance interne $C_{21} \rightarrow C_{22}$ de la figure 2.2 page 40 48
2.6 Transformation des dépendances sortantes $C_{11} \rightarrow D$ et $C_{22} \rightarrow D$ utilisant la même sortie de C dans la figure 2.2 page 40 49
2.7 Transformation du graphe flot de données conditionné de la figure 2.2 page 40 . . . 50
2.8 Exemple de graphe flot de données conditionné à deux niveaux de conditionnement 51
2.9 Mise à plat du graphe flot de données conditionné de la figure 2.8 page 51 52

3.1 FSM non hiérarchique et équivalent Statecharts 54
3.2 Éléments de Syntaxe SyncCharts 56
3.3 Exemple de programme SyncCharts : ABRO 58
3.4 Premier niveau hiérarchique de la traduction d'une constellation 59
3.5 Structure des sous-graphes appartenant à *CalculEtatCourant* 59
3.6 Structure d'une opération *FranchArcX* quand l'arc *x* est l'arc le moins prioritaire . 62
3.7 Structure d'un sous-graphe appartenant à *CalculEtatCourant* et correspondant à un état-puits 62
3.8 Exemple de traduction de composition parallèle 63
3.9 Exemple de SyncChart contenant un macro-état 64
3.10 Exemple d'opération FranchArcX traduisant le franchissement d'un arc de préemption forte entre deux macro-états 66
3.11 Deux exemples de programme SyncCharts utilisant des signaux locaux 67
3.12 Traduction du SyncChart de gauche de la figure 3.11 page 67 68
3.13 Exemple d'opération FranchArcX traduisant le franchissement d'une transition de préemption faible entre deux macro-états 69

3.14 Exemple de SyncChart contenant un macro-état pouvant être quitté par une terminaison normale . . . 70
3.15 Opération FranchArc1 appartenant à la traduction du SyncChart de la figure 3.14 page 70 . . . 71
3.16 SyncChart comportant des arcs instantanés et équivalent sans arc instantané . . . 73
3.17 Exemple de schéma-bloc Scicos . . . 76
3.18 Signal régulier dans Scicos et l'ensemble de périodes d'activation associées . . . 78
3.19 Exemples de multi-activation . . . 78
3.20 Utilisation de la rémanence sous Scicos pour obtenir la partie positive d'une sinusoïde 79
3.21 Même période ne veut pas dire synchronisme . . . 80
3.22 Exemple de sous-échantillonnage . . . 81
3.23 Exclusivité et synchronisme potentiel . . . 81
3.24 Utilisation de "Region to superbloc" sous Scicos . . . 82
3.25 La phase de pré-compilation Scicos . . . 83
3.26 Exemple de schéma-bloc dont tous les blocs ont des exécutions potentiellement synchrones et donc traduisible en graphe SynDEx . . . 87
3.27 Traduction d'un *if then else* en une opération conditionnante . . . 88
3.28 Traduction d'un bloc *if then else* et d'un bloc *sample and hold* en flot de données conditionné . . . 89
3.29 Traduction d'un bloc *if then else* et d'un bloc *sample and hold* multi-activé en flot de données conditionné . . . 89
3.30 Traduction des activations d'un schéma-bloc Scicos en hiérarchie d'opérations conditionnantes . . . 90
3.31 Traduction d'un schéma-bloc Scicos ne possédant qu'un signal d'activation . . . 90
3.32 Schéma-bloc Scicos et HOAA correspondant . . . 91
3.33 Cas $A_C \subset A_P$ . . . 93
3.34 Cas $A_P \cap A_C = \emptyset$, traduction erronée . . . 95
3.35 Cas $A_P \cap A_C = \emptyset$, traduction correcte . . . 96
3.36 Cas $A_P \cap A_C = \emptyset$ hiérarchique, traduction correcte . . . 96
3.37 Cas $A_P \subset A_C$, traduction erronée . . . 97
3.38 Cas $A_P \subset A_C$, traduction correcte . . . 98
3.39 Schéma-bloc correspondant au cas mixte . . . 99
3.40 Différentes étapes de traduction du schéma-bloc de la figure 3.39 page 99 . . . 100
3.41 Graphe SynDEx obtenu par traduction du schéma-bloc Scicos de la figure 3.32 page 91 . . . 101

4.1 Principes du logiciel SynDEx . . . 104
4.2 Exemple d'algorithme décrit avec le logiciel SynDEx . . . 105
4.3 Exemple d'architecture décrite avec le logiciel SynDEx . . . 105
4.4 Diagramme temporel résultant de l'adéquation de l'algorithme de la figure 4.2 avec l'architecture de la figure 4.3 . . . 106
4.5 Exemple d'algorithme conditionné décrit avec le logiciel SynDEx . . . 108
4.6 Mise à plat par le logiciel SynDEx de l'algorithme de la figure 4.5 page 108 . . . 112

4.7 Architecture décrite avec le logiciel SynDEx . . . 113
4.8 Diagramme temporel résultant de l'adéquation de l'algorithme de la figure 4.5 avec l'architecture de la figure 4.7 . . . 113

5.1 Chaîne de développement de la modélisation/simulation jusqu'à l'implantation . . 116
5.2 Spécification Scicos d'une application automobile couplant ABS et ESP . . . 117
5.3 Description hiérarchique du bloc ABS avec Scicos . . . 121
5.4 Programme SynDEx obtenu après traduction du bloc Scicos ABS . . . 122
5.5 Diagramme temporel résultant de l'adéquation de la traduction du bloc Scicos ABS avec une architecture constituée de deux processeurs reliés par un bus . . . 122
5.6 Algorithme du régulateur de vitesse : programme principal (Scicos) . . . 123
5.7 Programmes Scicos correspondant aux blocs *PedalsPressed* (à gauche) et *SpeedLimit* (à droite) . . . 123
5.8 Programme SyncCharts correspondant au bloc *SpeedFilter* . . . 124
5.9 Programme SyncCharts correspondant au bloc *CruiseState* . . . 124
5.10 Programme Scicos correspondant au bloc *CruiseSpeedMgt* . . . 125
5.11 Programme Scicos correspondant au bloc *Regulator* . . . 125
5.12 Programme SynDEx obtenu après traduction des programmes Scicos et SyncCharts de l'algorithme du contrôleur de vitesse . . . 126
5.13 Architecture spécifié avec SynDEx pour l'application de contrôleur de vitesse . . . 126
5.14 Diagramme temporel résultant de l'adéquation de l'algorithme du contrôleur de vitesse de la figure 5.12 avec l'architecture de la figure 5.13 . . . 127

6.1 Représentation graphique du modèle temps réel classique . . . 139
6.2 Preémption d'une tâche T_2 par T_1 . . . 141
6.3 Méthode d'ordonnancement en tâche de fond . . . 150
6.4 Serveur à scrutation . . . 152
6.5 Serveur ajournable . . . 153
6.6 Serveur à échange de priorités . . . 155
6.7 Serveur sporadique . . . 156
6.8 Serveur à échange de priorités dynamique . . . 159
6.9 Serveur sporadique dynamique . . . 161
6.10 Serveur à utilisation totale . . . 162
6.11 Earliest Deadline Late . . . 164
6.12 Exemple A pour la preuve de non-optimalité . . . 169
6.13 Exemple B pour la preuve de non-optimalité . . . 170
6.14 Exemple d'optimalité de la version améliorée du serveur à utilisation totale . . . 171
6.15 Temps de réponse moyens relativement à BS en fonction de la charge due aux tâches apériodiques . . . 172
6.16 Temps de réponse moyens en fonction de la charge due aux tâches apériodiques . . 173
6.17 Temps de réponse moyens relativement à BS en fonction de la charge due aux tâches apériodiques . . . 175

7.1 Modification de l'échéance de T_b pour contraindre sa date de fin d'exécution par rapport à T_a . . . 178
7.2 Exemple d'ordonnancement de T_1, T_2, T_3 respectant $\mathcal{C}$ . . . 180
7.3 Exemple d'ordonnancement satisfaisant des contraintes de latence . . . 186

8.1 Exemple de découpage d'ordonnancement en intervalles d'exécution . . . 192
8.2 Ordonnancement prévu à la fin du premier slot, après acceptation de la tâche apériodique ferme J_1 . . . 198
8.3 Ordonnancement prévu à la fin du septième slot, après acceptation de la tâche apériodique ferme J_3 . . . 199
8.4 Ordonnancement observé sur l'hyperpériode, avec exécution des tâches apériodiques fermes acceptées J_1 et J_3 ainsi que de la tâche apériodique souple J_4 . . . 200
8.5 Exemple d'ordonnancement possédant une échéance et une contrainte de latence . 201
8.6 Expression de la latence comme une échéance pour permettre le slot shifting . . . 202
8.7 Ordonnancement où $J_2(1,4,5)$ est acceptée et où les échéances et la contrainte de latence sont satisfaites . . . 203
8.8 Découpage de la tâche T_2 en deux tâches T_{2a} et T_{2b} appartenant à des intervalles d'exécution différents . . . 203
8.9 Apports de la thèse relativement à l'implantation distribuée des systèmes temps réel 226

Introduction

Contexte

Les systèmes temps réel

Cette thèse s'intéresse aux méthodes d'implantation des systèmes temps réel distribués. Par système* on entend système informatique interagissant avec l'environnement et non pas le système au sens des automaticiens qui inclut le plus souvent l'environnement. Il s'agit donc d'un ensemble de fonctionnalités implantées sur une architecture matérielle et satisfaisant des contraintes comportementales et/ou d'implantation. Lorsque les contraintes sont des contraintes temporelles qui imposent des dates limites à l'exécution de ces fonctionnalités, on qualifie le système de temps réel.

On désigne par systèmes réactifs* les systèmes qui doivent impérativement réagir aux stimuli de l'environnement en réalisant les fonctionnalités qui produisent des données agissant sur l'environnement [1]. Les fonctionnalités, ce que le système doit faire, sont décrites par des programmes constitués d'un ensemble d'instructions. S'il existe des possibilités de choix entre les instructions d'un programme (utilisation de structures comme if-then-else, gardes sur les transitions d'une machine à état), nous qualifions le programme de conditionné. Ainsi le conditionnement d'un programme désigne les différentes alternatives d'exécution qu'il offre.

Les systèmes temps réels sont de nos jours omniprésents et couvrent un large spectre d'applications. Des contraintes temporelles existent dans une automobile, un avion, mais aussi dans un téléphone portable ou un service de vidéos sur demande (VoD). Bien sûr ces contraintes ne sont pas aussi critiques d'un système à l'autre. Là où un délai dépassé pour la vidéo ne cause qu'un gel momentané de l'image sur l'écran du consommateur, le non-respect de la contrainte de temps sur le freinage d'une automobile peut entraîner des pertes humaines. On distingue donc habituellement le temps réel strict où toutes les contraintes doivent être respectées pour préserver l'intégrité du système et des utilisateurs, du temps réel souple (ou mou) où le non-respect des contraintes dégrade les performances du systèmes mais ne remet pas en cause son fonctionnement. Un système peut bien sûr posséder des fonctionnalités temps réel strict et d'autres temps réel souple [1].

Lors de l'implantation temps réel on substitue à une programme un ensemble de tâches auxquelles sont associées les contraintes temps réel. Ces tâches peuvent être périodiques ou apériodiques. Une tâche périodique est infiniment répétée à intervalles réguliers dont la durée est connue,

1. on écrit, ici et par la suite, "strict" et "souple" au singulier car on considère que ces adjectifs qualifient "temps réel" et non pas "fonctionnalités".

alors qu'une tâche apériodique demande son exécution à une date inconnue. Les tâches périodiques constituent le plus souvent la partie temps réel strict du système, alors qu'on ne peut associer aux tâches apériodiques que des contraintes temps réel souple, ces tâches étant imprévisibles.

Implantation distribuée

Ces systèmes sont de plus en plus souvent implantés sur des architectures distribuées, c'est-à-dire comptant plusieurs unités de calcul. Dans la suite du document, nous utilisons le terme "distribué" plutôt que "parallèle" ou "réparti" pour reprendre la terminologie anglo-saxonne usuelle de "distributed real-time". Bien que l'architecture matérielle puisse combiner processeurs (microcontrôleur, DSP) et circuits intégrés spécifiques (FPGA, CPLD), nous nous intéressons ici plus particulièrement aux unités de calcul de type processeur.

Il existe plusieurs raisons pour utiliser des architectures distribuées (multi-processeur) plutôt que centralisées (monoprocesseur). Tout d'abord, à puissance cumulée équivalente, cinq processeurs valent moins cher qu'un seul. On retrouve cette question de coût lorsqu'il faut remplacer un processeur défectueux. De plus, l'ajout de nouvelles fonctionnalités au système peut se faire par l'ajout d'un nouveau processeur, le système gagnant ainsi en modularité et en évolutivité. Enfin, dans un système où les capteurs et actionneurs sont nombreux, une architecture distribuée permet de réduire, par rapport à une architecture centralisée, les câblages en même temps que les perturbations électromagnétiques sur les signaux analogiques.

L'implantation distribuée d'un système consiste à distribuer les fonctionnalités décrites sous forme de programmes sur les unités de calcul de l'architecture, et pour chacune de ces unités à ordonnancer les fonctionnalités qui lui ont été allouées, c'est-à-dire trouver un ordre dans leur exécution.

L'approche hors-ligne

Nous nous intéressons aux systèmes possédant des contraintes temps réel strict et à l'implantation distribuée des fonctionnalités de ces systèmes. Les contraintes temps réel strict impliquent qu'on puisse assurer avant exécution que toutes les contraintes seront respectées quoiqu'il arrive. On distingue deux approches pour l'implantation temps réel, l'approche hors-ligne et l'approche en-ligne.

L'approche hors-ligne consiste à calculer une implantation avant l'exécution du système, de stocker l'ordonnancement et la distribution obtenus, puis à l'exécution de "lire" les informations stockées. L'avantage principal est qu'une fois l'ordonnancement et la distribution fixés, l'exécution possède un comportement totalement déterministe, ce qui simplifie le respect de contraintes temps réel strict. Un autre avantage est qu'il n'y a pas de limite à la complexité du calcul de cette implantation tant qu'un résultat peut être obtenu dans un temps raisonnable. Cela permet ainsi de prendre en compte des contraintes temporelles plus complexes ou le coût des communications entre unités de calcul [2][3].

Les inconvénients de l'approche hors-ligne sont d'une part qu'il n'est pas possible d'y prendre en compte des tâches au comportement non prévisible et d'autre part que le moindre ajout ou suppression de tâche nécessite un recalcul complet de l'ordonnancement et de la distribution.

L'approche en-ligne consiste à réaliser les choix de distribution et/ou d'ordonnancement lors de l'exécution en utilisant des algorithmes appropriés. Cette approche a tout d'abord l'avantage d'être flexible car il est simple d'ajouter ou supprimer une tâche ou d'en modifier les caractéristiques temporelles. D'autre part, puisque les choix sont effectués lors de l'exécution, il n'est pas nécessaire de connaître toutes les caractéristiques des tâches avant celle-ci, ce qui permet de prendre en compte des tâches au comportement non prévisible.

Un premier inconvénient de l'approche en-ligne est justement le non déterminisme impliqué par les décisions prises à l'exécution, car il est alors difficile de montrer que des contraintes temps réel strict sont respectées quoiqu'il arrive. Un deuxième inconvénient est que les algorithmes effectuant les choix doivent être de complexité faible afin d'en maîtriser le coût lors de l'exécution.

L'implantation distribuée temps réel strict nécessite de prendre en compte le coût des communications et de distinguer les différents processeurs et leurs caractéristiques (entrées/sorties, mémoires, vitesse de calcul). La complexité du calcul d'une implantation distribuée temps réel strict est de ce fait difficilement compatible avec une approche en-ligne. Dans le cas où le coût des communications et les différences entre processeurs sont pris en compte, il n'existe d'ailleurs pas d'algorithme optimal, c'est-à-dire permettant de toujours trouver, si elle existe, une implantation distribuée satisfaisant les contraintes. Dans ce cas, on doit employer hors-ligne des heuristiques pour trouver une implantation distribuée satisfaisante en faisant de l'exploration d'implantation.

Objectifs

Dans un contexte temps réel strict, l'utilisation de l'approche hors-ligne est donc nécessaire pour l'implantation de systèmes distribués. Relativement à l'approche hors-ligne, il nous semble intéressant de traiter de deux problèmes respectivement en amont et en aval de l'implantation des systèmes distribués :

- en amont, comment l'approche hors-ligne peut permettre la distribution de programmes conditionnés,
- en aval, comment exécuter en-ligne des tâches aux contraintes temps réel souple en respectant les contraintes temps réel strict des tâches ordonnancées hors-ligne.

Notre apport porte donc sur ces deux problématiques.

En amont ...

Dans les programmes conditionnés, les alternatives rendent l'implantation distribuée temps réel particulièrement complexe. Il faut en effet distribuer et ordonnancer conjointement les différents cas d'exécution en veillant à ce que chaque unité de calcul puisse choisir quelles instructions exécuter. Cela nécessite d'expliciter les relations entre les alternatives et les parties non conditionnées des programmes.

Si certains modèles de programmation permettent d'automatiser la distribution des différentes instructions d'un programme, les techniques existantes se limitent souvent aux programmes ne comportant pas d'alternatives ou, le cas échéant considèrent ces alternatives comme atomiques

[4], c'est-à-dire dont les instructions ne peuvent être exécutées sur des processeurs différents, ce qui nuit à l'efficacité de l'implantation distribuée obtenue.

Il est donc utile de proposer un modèle permettant non seulement la description de programmes conditionnés mais rendant également possible d'en obtenir l'implantation distribuée efficace. Les langages de programmation étant variés et souvent utilisés conjointement pour un même système, il est nécessaire de montrer comment traduire automatiquement des programmes utilisant ces langages en programme utilisant le modèle proposé. Il est alors possible d'utiliser ce modèle dans le processus de développement, après que le système ait été décrit par des programmes conditionnés, afin d'obtenir l'implantation distribuée efficace.

En aval ...

Une fois l'implantation distribuée calculée hors-ligne pour les fonctionnalités aux contraintes temps réel strict, il faut prendre en compte les fonctionnalités aux contraintes temps réel souple du système. En effet, s'il est possible de distribuer et ordonnancer hors-ligne des tâches périodiques, les tâches apériodiques ne peuvent être prises en compte qu'en-ligne où elles doivent être acceptées ou refusées. Il convient donc de permettre au mieux l'exécution des tâches apériodiques mais sans remettre en cause le respect des contraintes temps réel des tâches périodiques.

Il existe de nombreux travaux traitant de l'ordonnancement mixte de tâches périodiques et apériodiques [5], mais ils s'appliquent tous au modèle temps réel classique où la seule contrainte temps réel est l'échéance qui définit pour une tâche une date de fin d'exécution au plus tard. Cela ne permet pas d'imposer un délai entre l'exécution de deux tâches dépendantes, éventuellement par l'intermédiaire d'autres tâches. Or, cela peut être utile dans une loi de commande par exemple pour garantir que les données, produites par une tâche capteur, sont utilisées dans un délai borné par toutes les tâches chargées du calcul de l'asservissement. C'est pourquoi l'équipe AOSTE a introduit dans des travaux précédents la contrainte de latence qui permet d'imposer un délai entre la date de début d'exécution d'une tâche (appelée première tâche) et la date de fin d'exécution d'une autre (appelée deuxième tâche), ces tâches étant dépendantes.

La contrainte d'échéance définit pour une tâche une date de fin au plus tard qui reste fixe en-ligne, quoiqu'il arrive. Ce n'est pas le cas de la contrainte de latence qui définit pour la deuxième tâche une date de fin au plus tard qui varie en-ligne si la date de début d'exécution de la première tâche est pour une raison ou pour une autre, repoussée. La contrainte de latence offre ainsi plus de flexibilité en-ligne pour l'exécution de tâches apériodiques. Il convient alors de proposer des solutions d'ordonnancement mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques. Nous montrons que l'on peut, après ordonnancement hors-ligne, réduire les contraintes temps réel dont on doit assurer le respect pour que l'ensemble des contraintes soient, par implication, respectées. Nous donnons également des méthodes d'ordonnancement mixte adaptées dont nous discuterons l'efficacité.

Comme le résume la figure 1 page 19, nos objectifs concernent donc deux étapes de l'implantation distribuée des systèmes temps réel. La première étape (point d'interrogation de droite) correspond aux méthodes nécessaires pour obtenir une implantation distribuée de programmes conditionnés. La deuxième étape (point d'interrogation de la partie inférieure) consiste, quand à elle, à exécuter conjointement les fonctionnalités prévues hors-ligne avec d'autres fonctionnalités

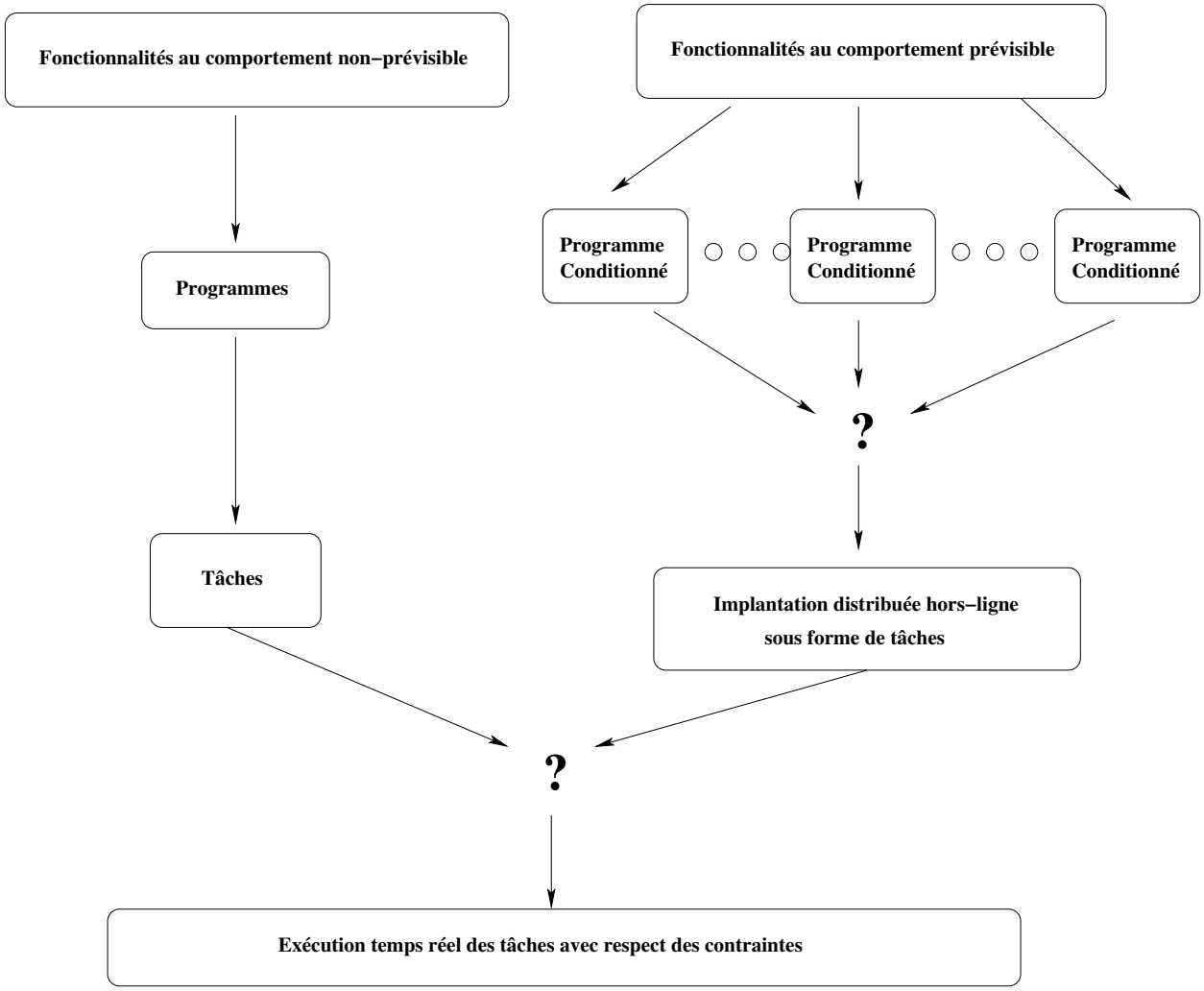

FIG. 1 – *Objectifs de la thèse relativement à l'implantation distribuée des systèmes temps réel*

imprévues, tout en respectant les contraintes temps réel. Les apports relatifs à ces deux problématiques doivent ainsi permettre de faciliter l'implantation des systèmes distribués temps réel.

Plan de la thèse

Dans la première partie de cette thèse nous proposons une méthode pour obtenir une implantation distribuée hors-ligne de programmes conditionnés. Après le chapitre 1 consacré à l'état de l'art des méthodes de description et d'implantation distribuée de programmes conditionnés, nous présentons dans le chapitre 2 un modèle flot de données conditionné bien adapté pour l'implantation distribuée. Ensuite, nous montrons dans le chapitre 3 comment des programmes conditionnés peuvent être traduits automatiquement en programmes utilisant ce modèle flot de données conditionné et donc mieux adapté à la distribution. Puis nous présentons dans le chapitre 4 le logiciel SynDEx qui permet d'obtenir automatiquement l'implantation distribuée de ces programmes flot de données conditionné. Nous illustrons dans le chapitre 5 notre apport par des exemples d'implantations distribuées de programmes conditionnés en utilisant les traductions automatiques et le logiciel SynDEx.

Dans la deuxième partie de la thèse, nous présentons un état de l'art de l'ordonnancement temps réel de tâches périodiques et apériodiques dans le chapitre 6. Nous introduisons ensuite au chapitre 7 une nouvelle contrainte temps réel, la latence, qui permet de contraindre la date limite de fin d'exécution d'une tâche par rapport au début de l'exécution d'une autre. Nous montrons son intérêt pour les ordonnancements hors-ligne de tâches périodiques puis nous décrivons dans le chapitre 8 comment, en étendant la méthode de slot shifting, il est possible d'ordonnancer en-ligne des tâches apériodiques tout en respectant les contraintes de latence satisfaites hors-ligne. Nous proposons ainsi deux méthodes que nous comparons en termes de complexité et d'optimalité.

Dans la suite du manuscrit, nous définirons de nombreux termes. Ces définitions pouvant être nouvelles pour le lecteur, il nous a semblé utile d'inclure un index à la fin du manuscrit. Afin de retrouver facilement la définition d'un terme, celui-ci est suivi sur la page indiquée d'une étoile, commme ceci*.

Première partie

Implantation distribuée de programmes conditionnés

Introduction de la partie I

Notre but est ici de proposer une méthode permettant d'obtenir une implantation distribuée de programmes conditionnés. Pour cela il convient d'abord de décrire comment des alternatives de fonctionnalités d'un système sont décrites par des programmes conditionnés. Nous verrons que l'expression de ces alternatives peut prendre plusieurs formes.

L'état de l'art montrera que les principales méthodes de développement des systèmes distribuées consistent à représenter le système distribué comme la coopération de plusieurs systèmes monoprocesseur. Ces méthodes sont sources d'erreurs car elles nécessitent énormément de décisions empiriques de la part du ou des développeurs. Ils doivent ainsi décider du découpage des fonctionnalités pour obtenir ces différents systèmes monoprocesseur. Ensuite, ils doivent construire les communications inter-processeurs découlant du découpage des fonctionnalités. Les erreurs que ces décisions engendrent sont en outre très difficiles à résoudre car il n'existe pas réellement de lien entre les différentes phases de développement (description des fonctionnalités, découpage, construction des communications). Ainsi, la prise en compte des alternatives de fonctionnalités lors de la distribution est réduite à néant et les développeurs préfèrent centraliser les différentes alternatives sur un seul processeur de manière à ne pas compliquer la construction des communications.

Nous verrons ensuite que les langages les plus adaptés à l'implantation distribuée sont basés sur le modèle flot de données. Néanmoins nous montrerons que s'ils permettent de décrire des programmes conditionnés, ils ne permettent pas d'en obtenir une implantation distribuée où les alternatives de fonctionnalités sont réellement distribuées. C'est pourquoi nous proposerons un modèle flot de données conditionné permettant la distribution de ces alternatives.

Nous présenterons également des traductions de langages, permettant le conditionnement, en langage basé sur notre modèle afin de rendre possible l'implantation distribuée de programmes utilisant des langages différents. Nous illustrerons ces traductions par des exemples, en utilisant le logiciel SynDEx qui permet l'implantation distribuée de programmes flot de données conditionnés.

Chapitre 1

État de l'art

1.1 Concepts de base

Un algorithme*, pour Turing [6], est une "séquence finie d'opérations" (réalisables en un temps fini et avec un nombre de ressources fini). Ici, on étend cette notion d'algorithme afin de prendre en compte le caractère infiniment répétitif des systèmes réactifs que sont les systèmes distribués temps réel nous intéressant. En effet, le nombre d'interactions effectuées par un système réactif avec son environnement n'est pas borné a priori. C'est cette notion étendue du terme algorithme que nous utiliserons par la suite.

Afin d'être implanté sur une architecture matérielle pour constituer un système, les fonctionnalités sont décrites sous la forme d'un algorithme constitué d'un ensemble de programmes*. Un programme peut être vu soit comme un ensemble de traitements atomiques (ce dont l'algorithme a besoin) ou comme une suite de traitements atomiques (comment ceux-ci sont organisés). Ceci est résumé par l'équation de Kowalski " Algorithm = logic + control" [7], le logic désignant le "quoi" et le contrôle le "comment". Ici, le terme atomique* s'applique à ce qui ne peut être divisé. Notre but étant l'implantation distribuée, un traitement atomique ne pourra donc pas être découpé en deux traitements s'exécutant sur deux processeurs différent. Plus généralement, un traitement atomique désigne un traitement de base, ne pouvant pas être décomposé en un ensemble de traitements plus simples.

Considérons l'exemple d'un algorithme devant faire la somme de quatre entiers. Le traitement atomique (la partie "logique" de l'équation) est l'addition de deux entiers. Il faut ensuite préciser le contrôle, c'est-à-dire comment obtenir le résultat (la somme de quatre entiers) en utilisant ce traitement atomique. Il est possible avec les mêmes traitements atomiques d'obtenir deux algorithmes équivalents, c'est-à-dire réalisant le même traitement, en ne modifiant que le contrôle. On peut par exemple réaliser la somme des deux premiers entiers, puis la somme du résultat et du troisième entier et enfin la somme du résultat et du quatrième. Mais on peut aussi réaliser la somme des deux premiers entiers, puis celle des deux derniers et ensuite additionner les deux résultats.

En précisant le contrôle d'un algorithme, on permet son implantation sur une machine de Von Neumann* [8], qui est le modèle sur lequel sont basés les processeurs actuels. Une machine de Von Neumann possède un état correspondant au contenu de sa mémoire et de certains registres

du processeur (dont le compteur programme). La machine exécute les instructions (correspondant aux opérations de Turing) les unes après les autres dans un ordre précis et le résultat de chaque instruction est que le contenu d'une ou de plusieurs adresses mémoire (ou registre) change de valeur.

Le contrôle* consiste alors à définir :

- une séquence, c'est-à-dire un ordre, entre les traitements atomiques. Cette séquence peut utiliser plusieurs fois les mêmes traitements atomiques éventuellement eux-mêmes mis en séquence (boucles) ;
- les états mémoires correspondant à cette séquence ;
- les choix, c'est-à-dire les alternatives possibles entre plusieurs traitements atomiques au sein d'une séquence. Ce choix étant réalisé en fonction du résultat d'un test, on parlera par la suite de "conditionnement".

On parle dans ce qui suit de modèles de programmation* là où Floyd [9] parle de "paradigmes de programmation" de haut niveau, pour désigner les paradigmes permettant la description générales des algorithmes, par oppositions aux paradigmes de programmation bas niveaux (système) comme ceux concernant la gestion des données (gestion de copies versus données partagées) ou les communications (mode connecté versus non connecté). Les modèles de programmation sont donc pour nous un ensemble de concepts liés à la description des systèmes.

Un langage de programmation* applique les concepts d'un ou plusieurs modèles et permet de décrire un algorithme sous forme d'un programme.

Par la suite, ce que nous désignerons par langages de programmation rejoindra parfois ce que le lecteur considère comme étant des langages de spécification. Nous choisissons programmation dans le sens où pour ces langages il existe une façon automatique (parfois via des traductions en d'autres langages) d'obtenir un code exécutable à partir de la description. Nous étendons en ce sens la notion habituelle de langage de programmation limitée habituellement aux langages qui possèdent un compilateur permettant d'obtenir directement du langage machine.

Notre volonté est d'automatiser l'obtention de l'implantation distribuée à partir de programmes de manière à réduire les interventions et interprétations humaines. Il est donc utile de s'assurer que les programmes initiaux sont corrects et pour cela ils doivent utiliser des langages formelles. Ainsi, nous ferons souvent référence par la suite aux langages synchrones* qui répondent bien aux besoins des systèmes réactifs temps réel[1][10]. Le temps y est modélisé comme une suite d'instants logiques* définissant une horloge de base et l'hypothèse synchrone implique que les réactions sont produites au même instant que les événements qui les engendrent. Cela permet de se détacher du temps d'exécution, rendant la description de l'application indépendante de l'architecture matérielle. On dénombre à ce jour trois langages synchrones majeurs, Signal [11], Esterel [12] et Lustre [13]. Ces langages permettent d'effectuer des vérifications de propriétés, que ce soit sur des programmes Lustre[14], Esterel[15] ou Signal[16]. Pour un panorama des travaux réalisés lors des dix dernières années autour des langages synchrones, le lecteur pourra consulter [17].

1.2 Deux approches différentes pour décrire un algorithme

Lorsqu'un algorithme s'exécute sur un processeur via un programme, c'est que son contrôle a été intégralement défini, de manière à correspondre à celui que nécessite une machine de Von Neumann. Néanmoins, le développeur peut, en utilisant un langage où le contrôle est implicite, ne pas avoir décrit si précisément le contrôle. Dans ce cas, le programme sera ensuite transformé en langage machine de Von Neumann par une compilation qui précisera le contrôle.

On discerne ainsi deux approches dans la description des algorithmes que les auteurs de [18] appellent "operational" et "definitional". L'approche "operational" consiste à décrire l'algorithme par un programme proche du modèle de la machine de Von Neumann. Il s'agit alors de décrire les instructions et les états successifs du système menant de l'état initial à l'état final correspondant au résultat souhaité. On peut citer comme appartenant à cette approche le modèle impératif [19] et le modèle objet [20]. L'approche "definitional" (dite aussi déclarative) consiste quant à elle à définir le résultat d'un algorithme par un ensemble de propriétés plutôt que de s'attacher à décrire comment l'obtenir. Implicitement le contrôle dépend alors des relations entre ces propriétés. Le modèle fonctionnel [21] tel qu'il est appliqué dans Haskell [22] ou dans une partie de CAML [23] appartient à cet approche.

1.3 Modèle flot de contrôle

Le modèle flot de contrôle* correspond à l'approche "operational" et consiste à expliciter principalement le contrôle d'un algorithme en représentant la séquence et le conditionnement des traitements atomiques ainsi que les états associés. Les relations entre les traitements atomiques, c'est-à-dire les données qu'ils manipulent, ne sont quant à elles pas décrites, ce modèle utilisant des variables partagées pour la représentation des données.

Les langages textuels impératifs comme le langage C utilisent le modèle flot de contrôle. Le ";" spécifie la séquence, l'instruction "for" les boucles et l'instruction "if() then () else ()" le conditionnement. L'état est alors explicite à travers des variables dont dépendent les choix des traitements atomiques (conditionnement) et implicite dans le compteur de programme. Les variables sont partagées et peuvent être modifiées par n'importe quel traitement atomique.

Fréquemment utilisé en amont de la description d'un algorithme, l'organigramme (*programm chart* en anglais) représente le flot de contrôle sous forme de graphe. Chaque sommet est un traitement atomique ou un test (conditionnement) et chaque arc défini un ordre d'exécution entre les traitements atomiques (séquence). L'état est implicite et correspond aux valeurs contenues dans les variables partagées par chaque nœud.

Enfin les machines à états finis* (FSM* pour *Finite State Machine*) sont des graphes explicitant les états. Chaque sommet est un état et chaque arc, appelé transition, définit une condition pour passer d'un état à un autre (conditionnement). Les traitements atomiques sont soit attachés aux transitions (machines de Mealy [24]), soit aux états (machines de Moore [25]). Les arcs définissent donc une séquence entre les traitements. Les traitements atomiques peuvent effectuer des lectures et des écritures dans des variables partagées par l'ensemble du graphe.

L'intérêt des machines à états finis consiste justement dans le fait qu'elles font apparaître un

nombre fini d'états. Il est ainsi possible de vérifier certaines propriétés (Peut-on atteindre tel état à partir de n'importe quel état ? Est-on sûr de ne pouvoir être dans tel état sans être passé d'abord par tel autre ?) qu'il serait plus difficile de vérifier avec un programme C par exemple dans lequel le nombre d'états n'est pas explicite [26].

Les langages Grafcet [27] [28] et Statecharts* [29] sont des exemples de langages utilisant le modèle flot de contrôle. Dans un programme utilisant le modèle flot de contrôle, l'absence de relations entre les traitements atomiques partageant une même variable peut rapidement amener à ce que l'état final des variables diffèrent de celui souhaité. Néanmoins, la possibilité de décrire précisément le contrôle de l'algorithme permet, sur certaines parties de l'algorithme, d'en optimiser le fonctionnement que ce soit au niveau de la rapidité ou du coût mémoire. Le langage Esterel* [12][30] développé à l'INRIA de Sophia Antipolis et ENSMP (École Nationale Supérieure des Mines de Paris), est le plus ancien des langages synchrones. Langage impératif textuel utilisant le modèle flot de contrôle, il permet de décrire l'exécution parallèle, la séquence d'actions et l'attente d'évènements, ce qui en fait un langage adapté pour la description textuelle des FSM. Il existe également des langages synchrones permettant la description graphique de FSM comme Argos* [31] ou SyncCharts* [32], ce dernier pouvant être considéré comme la version graphique d'Esterel.

La société Esterel Technologies [1] distribue Esterel Studio, version commerciale d'Esterel et le logiciel Scade où il est possible de décrire des FSM appelées "Safe State Machine" en utilisant le langage SyncCharts.

1.4 Modèle flot de données

Correspondant à l'approche "definitional", le modèle flot de données* tend à définir le résultat de l'algorithme par un ensemble de traitements atomiques et de relations entre ces traitements. La notion de flot vient du fait que l'algorithme est alors représenté comme un réseau de canalisations véhiculant un flux de données. Chaque nœud (raccordement entre les tuyaux) consomme le ou les flux des données entrants et produit le ou les flux de données sortants. Ainsi la description d'un algorithme avec le modèle flot de données nécessite de définir les traitements atomiques comme les nœuds de ce réseau et ensuite de préciser les tuyaux raccordant ces nœuds. La représentation graphique est alors naturelle, chaque sommet (raccordement) appelé opération* est un traitement atomique et chaque arc (tuyau) est une dépendance de données* entre deux opérations telle que la source produit la donnée nécessaire à la destination qui la consomme. Pour une dépendance de donnée, il ne peut y avoir qu'une seule opération source, c'est-à-dire pouvant produire cette donnée (pas de confusion), mais il peut y avoir plusieurs opérations destination (diffusion, l'arc devenant alors un hyperarc), c'est-à-dire pouvant consommer cette donnée. Une donnée n'est alors partagée que par son producteur et son ou ses consommateurs, tels que définis par la dépendance de données. Elle ne peut donc pas être utilisée par d'autres opérations que celles-ci.

On trouve la première référence au concept de flux de données dans l'article [33] de Karp et Miller lorsqu'ils décrivent leurs "computation graphs". Néanmoins le modèle flot de données n'est défini que quelques années plus tard par Dennis [34]. Celui-ci précise notamment la règle

1. http://www.esterel-technologies.com

d'activation* d'une opération dans un programme flot de données :

- une opération ne peut être exécutée que si toutes ses données d'entrée sont disponibles (une donnée sur chaque arc entrant) et que toutes celles de sorties ont été consommées ;
- une opération qui s'exécute consomme une donnée sur chacune de ses dépendances de données entrantes et produit une donnée sur chacune de ses dépendances de données sortantes.

Une dépendance de données définit donc un ordre d'exécution entre deux opérations. Au contraire, en absence de dépendance de données entre deux opérations il n'y a pas d'ordre imposé. L'ensemble des dépendances de données ne définit donc pas une séquence (ordre total) entre les opérations mais uniquement un ordre partiel. Une séquence de traitements atomiques pour une exécution sur une machine de Van Neumann n'est donc pas explicité lors de la programmation.

On appelle prédécesseur direct* d'une opération *A* une opération qui est la source d'une dépendance de données dont l'opération *A* est la destination. On appelle prédécesseur* d'une opération *A* toute opération telle qu'il existe un chemin [2] allant de cette opération à l'opération *A*. De même, on appelle successeur direct* d'une opération *A* une opération qui est la destination d'une dépendance de données dont l'opération *A* est la source. On appelle successeur* d'une opération *A* toute opération telle qu'il existe un chemin allant de l'opération *A* à cette opération.

Afin de décrire les algorithmes des systèmes réactifs, le modèle flot de données de Dennis a été étendu à l'infini [35]. Le graphe flot de données alors infini est la répétition infinie d'un sous-graphe motif. Pour des questions de simplicité on ne représente que ce sous-graphe motif qui correspond à une réaction du système. Les données produites lors d'une répétition de ce motif ne sont plus disponibles lors de la suivante, et dans le cas où une opération nécessite lors de sa $n^{ème}$ exécution de consommer une donnée produite à la $(n-1)^{ème}$ exécution d'une autre opération, on connecte entre ces deux opérations une opération appelée *retard**. Cette opération consomme une donnée sur son arc d'entrée après avoir produit sur son arc de sortie la donnée lue sur son arc d'entrée lors de son exécution précédente. Ces sommets *retard* permettent ainsi de mémoriser d'une répétition à l'autre l'état du graphe. Une opération sans arc d'entrée (respectivement de sortie) représente une interface d'entrée (respectivement de sortie), par exemple un capteur (respectivement un actionneur), avec l'environnement physique.

Décrire du conditionnement avec le modèle flot de données n'est pas naturel car il remet en cause la règle d'activation. En effet, s'il y a deux opérations représentant deux alternatives, suivant le choix effectué, seule l'une d'elle est exécutée et produit des données. Cela implique alors que les successeurs directs de l'opération non exécutée ne soient pas exécutées à cause de la règle d'activation. Il est donc impossible de mettre en dépendance des opérations correspondant à des alternatives et des opérations n'en étant pas. On définit donc deux catégories de programmes flot de données, les programmes flot de données homogènes* (aussi appelés réguliers) et les programmes flot de données hétérogènes*. Les premiers sont ceux respectant la règle d'activation donnée précédemment et qui ne permettent pas de décrire du conditionnement : toutes les opérations d'un programme sont exécutées, consomment toutes leurs entrées et produisent toutes leurs sorties. Dans les programmes flot de données hétérogènes des opérations peuvent ne pas consommer toutes leurs données en entrée et/ou ne pas produire toutes leurs sorties. Cela permet alors de définir des opéra-

2. un chemin menant de A à B est une suite alternée d'arcs et de sommets permettant d'aller de A à B en suivant l'orientation des arcs, au contraire d'une chaîne où l'orientation des arcs n'est pas prise en compte.

tions spéciales (comme les opérations *switch* et *merge* de Dennis) s'intercalant entre les opérations définissant des alternatives et celles n'en étant pas.

De manière générale les langages utilisant le modèle flot de données définissent différentes méthodes de spécification du conditionnement, allant de l'ajout d'une entrée booléenne permettant de choisir d'exécuter ou non l'opération [36], jusqu'à la possibilité de décrire plusieurs alternatives d'opérations à l'aide d'opérations spéciales et de choisir quelle alternative exécuter en fonction de la valeur d'une donnée [37].

Les deux premiers langages flot de données furent VAL [38] devenu ensuite SISAL et Id [39] dont le successeur est Id Nouveau [40]. Ce modèle est souvent utilisé dans les systèmes parallèles et des machines différentes des machines de Von Neumann ont été développées sur ce modèle [40]. Comme langages fondés sur le modèle flot de données, on peut citer Simulink qui repose sur Matlab [3], ou encore LabVIEW [4]. On parlera par la suite de langages flot de données pour désigner un langage basé sur le modèle flot de données.

Le langage Lustre [*][13][41] développé à l'IMAG (Institut de Mathématiques Appliquées de Grenoble), est un langage synchrone flot de données. Un programme se compose d'équations (au sens mathématique) et d'assertions (elles spécifient des propriétés du programmes en vue de vérifications). Cela permet ainsi de définir les opérations effectuées sur des flots (séquence de valeurs associées à une horloge dérivée de l'horloge de base). Le langage Signal[*] [11][42] développé par l'INRIA de Rennes et l'IRISA (Institut de Recherche en Informatique et Système Aléatoire), est très proche de Lustre. Manipulation de valeurs et horloges de flots sont leurs points communs, mais en Signal contrairement à Lustre, il n'existe pas d'horloge de base, elle est déduite à partir des relations exprimées dans le programme. Il existe également des versions graphiques des langages Lustre et Signal. Ces deux langages synchrones sont commercialisés, par TNI qui distribue l'outil SILDEX, basé sur le langage Signal, et par Esterel Technologies qui distribue Scade, la version commerciale de Lustre.

1.5 Différences entre les deux modèles

Un premier point de divergence entre les deux modèles est la gestion des données. Dans un graphe flot de données, chaque donnée est spécifiée par un arc. Si une opération manipule des données qui lui sont propres (non spécifiées par un arc [5]), celles-ci ne peuvent être visibles (en écriture et en lecture) par une autre. Au contraire, dans un programme flot de contrôle, les données sont des variables partagées et plusieurs traitements atomiques peuvent modifier les mêmes données. Dans une FSM, transitions et traitements atomiques peuvent également partager les mêmes données.

Le deuxième point de divergence est le parallélisme offert par les programmes utilisant ces deux modèles. Nous avons vu que dans un graphe flot de données, les arcs ne définissent qu'un ordre partiel sur les opérations. Il existe donc du parallélisme entre deux opérations n'ayant pas de relation d'ordre entre elle. Nous appellerons ce parallélisme "parallélisme potentiel[*]" [35] par

3. www.mathworks.com

4. www.ni.com/labview/

5. Ce qui est possible, une opération correspondant à une traitement atomique pouvant avoir des variables internes, mais pas conseillé car le modèle consiste justement à exprimer les dépendances de données.

opposition au parallélisme physique qu'offre réellement l'architecture matérielle. Dans un langage utilisant le modèle flot de contrôle, une implantation distribuée nécessite le plus souvent d'étudier le contrôle du programme pour "casser" la séquence décrite, en portions de programmes pouvant s'exécuter en parallèle. Cette étude nécessite de faire apparaître a posteriori l'utilisation des variables (écriture/lecture) par les différents traitements atomique, ce qui revient à construire les dépendances de données liant ces traitements. On peut aussi composer plusieurs programmes séquentiels en parallèle comme dans CSP [43], les différents programmes peuvent communiquer par passage de messages qui doivent être explicités lors de la programmation. Dans une FSM ou un organigramme, l'ordre impliqué par les arcs est total (séquence). Néanmoins il est possible, dans la plupart des langages utilisant les FSM, de décrire plusieurs FSM en parallèle, ce qui est appelé composition. Mais l'absence de la spécification explicite des dépendances de données empêche l'exploitation directe de ce parallélisme. Sur la figure 1.1 page 31, se trouve un exemple de com-

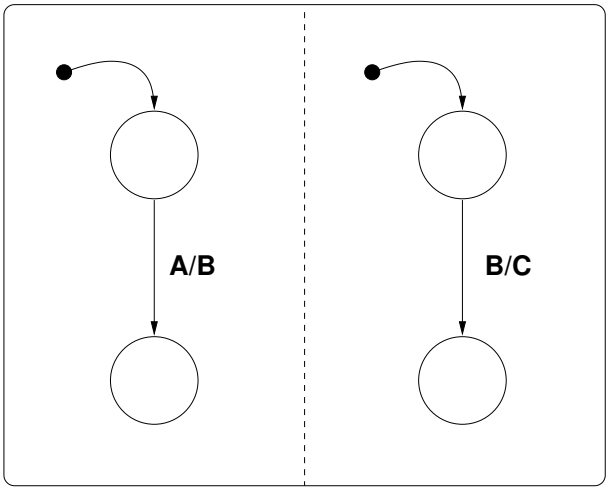

FIG. 1.1 – *Exemple de description de parallélisme avec des FSM*

position de FSM dans un graphe Statecharts, symbolisée par les pointillés séparant les deux FSM. Les transitions sont marquées sous la forme X/Y où X est la donnée devant être produite pour franchir la transition, et Y est la donnée produite par ce franchissement. En regardant plus attentivement le comportement des deux FSM, on s'aperçoit que le premier produit une donnée attendue par le deuxième. En effet, le premier attend A puis produit B alors que le deuxième attend B et produit C. Ainsi il existe implicitement une dépendance de données entre les deux FSM rendant impossible l'exploitation du parallélisme décrit. Ce dernier n'est que du parallélisme logique aussi appelé concurrence.

Ainsi, dans un programme flot de contrôle, le parallélisme potentiel ne peut être exhibé qu'après l'étude des dépendances de données existant implicitement à travers les variables utilisées dans un même programme (composition de FSM) ou dans la composition de plusieurs (CSP). Par contre, si l'algorithme est décrit par un programme flot de données, le parallélisme potentiel ainsi que les dépendances de données peuvent être utilisés pour obtenir une implantation distribuée sans à avoir à résoudre les problèmes dûs à l'emploi de variables globales.

Il existe en outre une différence entre les langages flot de données et les langages FSM sur la façon dont le graphe représente le caractère réactif du système. Considérons la $n^{ème}$ réaction du système. Si l'algorithme utilise un modèle flot de données, l'ensemble du sous-graphe motif est exécuté (sauf si conditionnement pour certaines opérations). On a donc utilisation de toutes les données d'entrée et production de toutes les données de sortie. Si l'algorithme est au contraire décrit par une FSM, cela dépend de l'état courant au début de la réaction du système. Partant de cet état, il ne pourra être franchi qu'un certain nombre de transitions dans la FSM [6] et il n'existe donc qu'un nombre restreint d'états de la FSM qui soient atteignables lors de cette réaction du système. Correspondant à ce sous-ensemble d'états, il n'y a qu'un sous-ensemble de transitions devant être évalué et qu'un sous-ensemble des opérations pouvant être exécuté. Ainsi l'ensemble des données d'entrée n'est pas forcément nécessaire, de même que l'ensemble des données de sortie n'est pas forcément produit.

Il apparaît donc qu'une distribution de programmes est plus facile avec un programme flot de données, puisque le parallélisme potentiel apparaît directement dans le graphe et que les dépendances de données empêchent les problèmes de cohérence de données inhérent à la distribution des variables partagées. Cependant, le flot de contrôle permet de maîtriser parfaitement l'ordre des traitements atomiques et les différents états, ce qui peut être important pour certaines parties d'un algorithme. De plus, le flot de contrôle permet, via les FSM, une représentation très répandue du conditionnement alors qu'il n'existe pas de standard dans le flot de données.

1.6 Utilisation conjointe des deux modèles

Nous avons donc deux modèles de programmation distincts, l'un mettant l'accent sur l'ordre des traitements atomiques menant à un résultat, l'autre sur les traitements atomiques à réaliser et les dépendances de données entre ces traitements. Suivant les fonctionnalités à décrire, il pourra être intéressant d'utiliser l'un ou l'autre de ces modèles. Ainsi, pour décrire un protocole de communication (synchronisations, émissions, réceptions) nécessitant de lier les différents traitements atomiques à des interruptions systèmes, il sera préférable d'utiliser un langage flot de contrôle et pourquoi pas une FSM. Par contre, la description d'une loi de commande, où les données sont traitées par différentes fonctions (gains, PID), sera plus aisée avec un langage flot de données. La complexité grandissante des fonctionnalités des systèmes nous intéressant mène souvent à décrire l'algorithme par plusieurs programmes, en utilisant pour certains un langage flot de contrôle et pour d'autres un langage flot de données. À un algorithme correspond donc un ensemble de programmes utilisant des langages basés sur différents modèles. La plupart des outils de développement intégrant ces langages permettent de simuler, sur la machine de développement, l'exécution du programme et, dans le cas des langages synchrones par exemple, de faire en plus de la vérification sur certaines propriétés de ce programme. Se pose ensuite le problème de l'implantation distribuée de ces programmes utilisant des langages différents.

6. Une seule transition dans la plupart des langages, mais certains permettent l'emploi de transitions et d'états "instantanés". Il est alors possible, au cours d'une seule réaction du système, de traverser plusieurs états. Leur emploi reste néanmoins marginal car peut mener à des FSM instables, c'est-à-dire dans lesquelles il est impossible de déterminer l'état du système en fin de réaction.

1.7 Implantation distribuée de programmes flot de contrôle et flot de données

1.7.1 Implantation distribuée de programmes séparés

La première approche consiste à tenter d'implanter séparément les différents programmes constituant l'algorithme. Pour chacun des langages employés, on peut habituellement générer du code monoprocesseur, et chaque programme devient, après compilation, un code séquentiel que les développeurs doivent allouer à un des processeurs de l'architecture distribuée. Puisque certaines données sont communes ou échangées par les différents programmes, les développeurs doivent ensuite distribuer et ordonnancer les communications entre les processeurs. En plus des erreurs humaines dans l'ajout des communications, l'implantation à partir de codes obtenus séparément conduit généralement à un comportement d'ensemble différent de celui souhaité, et cela même sur une architecture monoprocesseur, en absence de communications [44][45].

De plus, la distribution à la main de programme nécessite de réaliser des choix de distribution difficiles. En effet, dans les systèmes réactifs les entrées correspondent à des capteurs et les sorties à des actionneurs. Or, avant même l'implantation, les processeurs de l'architecture auxquels sont connectés ces capteurs et ces actionneurs sont souvent connus et imposés, que ce soit pour minimiser le câblage ou pour réduire les perturbations électromagnétiques. Il faudra donc rajouter des communications entre le processeur auquel sera alloué un code et les processeurs auxquels sont connectés les capteurs et actionneurs utilisés par le programme correspondant. Le nombre des communications à rajouter différant selon le choix du processeur auquel on alloue le code, choisir le bon processeur n'est pas facile.

D'autre part nous avons vu que dans le cas d'une FSM, l'ensemble des entrées n'est pas forcément nécessaire au même titre que l'ensemble des sorties n'est pas forcément produit. Dans le cas où un code monoprocesseur correspondant à une FSM est alloué à l'un des processeurs, il n'existe pas d'autre solution que de renseigner, lors de chaque réaction du système, l'ensemble des entrées et produire l'ensemble des sorties. Ce qui entraîne encore le rajout de communications si ces entrées et sorties correspondent à des capteurs et actionneurs connectés à d'autres processeurs.

Face à ces problèmes une autre approche consisterait à ne plus considérer l'implantation de programmes séparés mais l'implantation distribuée d'un seul programme qui serait l'unification de tous les autres. Il serait alors possible de générer automatiquement les communications découlant de la distribution de ce programme. Cette unification de programmes est rendu possible par l'inclusion de programmes dans un autre ou par des traductions permettant de transformer un programme utilisant un langage en un programme utilisant un langage différent. L'unification de programmes est le thème de la sous-section suivante. Ensuite nous verrons les travaux portant sur la distribution d'un programme unifié.

1.7.2 Unification de programmes

Il existe tout d'abord des logiciels de développement qui réunissent plusieurs langages de programmation, le plus souvent un langage flot de contrôle et un langage flot de données. C'est le

cas par exemple lorsque le langage flot de données Simulink est utilisé avec le langage flot de contrôle Stateflow (très proche de Statecharts). Il est ainsi possible dans le programme flot de données Simulink de décrire un sommet par une FSM Stateflow. De manière similaire, le logiciel Scade permet de décrire le sommet d'un programme flot de données Lustre par un programme flot de contrôle SyncCharts (renommé pour l'occasion Safe State Machine). Scade permet également d'importer un programme Simulink sous la forme d'un programme flot de données Lustre. Les automates de mode [46] permettent également de décrire par des FSM les sommets d'un programme flot de données et de décrire un état d'une FSM par un programme flot de données. Néanmoins les restrictions de sémantiques sur les programmes flot de données (au niveau des horloges) et sur les FSM (machines de Moore seulement) sont importantes. Enfin, on peut citer Ptolemy [47][4] un environnement de programmation hétérogène permettant de décrire n'importe quel sommet d'un programme flot de données par une FSM et inversement [26]. Il existe de nombreuses autres associations de langages flot de données et de flot de contrôle mais pour lesquels ne sont pas définies des règles d'unification, le lecteur pourra néanmoins consulter l'article [26] pour d'autres références.

Il existe aussi des traductions de langages où un programme flot de contrôle est transformé en un programme flot de données. C'est le cas des langages flot de contrôle Argos et Statecharts dont les programmes peuvent être traduits en programmes flot de données Signal [48][49]. Les principes de cette traduction de Statecharts vers Signal sont repris dans le logiciel Sildex qui permet d'importer des programmes Simulink et Stateflow sous la forme de programmes flot de données Signal.

Notre soucis étant l'implantation distribuée des programmes, il est intéressant de préciser la granularité des programmes obtenus après unification. Par granularité nous entendons le découpage du programme en traitements atomiques qui pourront être distribués sur l'architecture. Dans un programme flot de données homogène, la granularité correspond aux sommets du graphe correspondant au programme. Ainsi, les unifications consistant en une inclusion de programme flot de contrôle dans un sommet du programme flot de données amènent à considérer le programme flot de contrôle en entier comme un seul grain, c'est-à-dire un traitement atomique, et ce quelle que soit la concurrence décrite dans ce programme. Il en est ainsi dans les logiciels Scade et Simulink avec respectivement les programmes SyncCharts et Stateflow. Pour Stateflow, le programme devient une fonction C compilée que le programme Simulink exécute. Pour SyncCharts, le programme est lui aussi compilé en une fonction C à l'aide du compilateur d'Esterel. Néanmoins une prochaine version de Scade permettra de traduire le programme SyncCharts en programme Lustre et d'inclure celui-ci dans le programme Lustre principal [50]. En ce qui concerne Ptolemy, il n'y a pas réellement d'unification, la granularité dépendant alors du langage employé. Un programme flot de contrôle n'y est considéré que comme un seul grain. Dans les programmes flot de données, les sommets définissant des alternatives différentes (conditionnement) sont regroupés pour former un seul grain du point de vue de la distribution, transformant ainsi des programmes flot de données hétérogènes en programmes flot de données homogènes.

1.7.3 Implantation distribuée d'un programme flot de données

Nous ne prétendons pas ici faire un panorama exhaustif des méthodes de distribution de programme. Nous nous contentons de celles s'appliquant aux modèles cités précédemment. De même, bien qu'il existe de nombreux travaux sur la distribution de programme flot de contrôle (les programmes C par exemple), ces méthodes reposent sur une étude des dépendances de données entre les traitements atomiques et reviennent donc à traduire du flot de contrôle en flot de données. C'est pourquoi nous nous restreignons à la distribution de programme flot de données.

Dans [51] SynDEx (que nous présenterons plus loin) est utilisé pour distribuer des programmes flot de données Signal. Dans [52] un programme séquentiel est dupliqué sur les processeurs de l'architecture distribuée et chacun en exécute une partie. Les auteurs de cet article présentent des algorithmes permettant de reconstruire les dépendances de données afin de rajouter automatiquement les primitives d'envoi et de réception de données. Ces ajouts se font après distribution, ce qui signifie que les communications, et notamment les durées de celles-ci, ne sont pas prises en compte lors de la distribution et de l'ordonnancement. Dans [53] la description de l'algorithme se fait sous Simulink qui permet la simulation, puis le programme est traduit en un programme Scade (Lustre) pour validation, enfin le programme est implanté sur une architecture distribuée de type TTA (Time-Triggered Architecture). Toute la chaîne est automatisée, les auteurs présentant une transformation Simulink/Scade, ainsi qu'une extension de Scade pour générer du code temps réel distribué incluant les communications inter-processeurs.

1.8 Conclusion de l'état de l'art

Il existe donc des méthodes pour inclure des programmes flot de contrôle dans des programmes flot de données et les distribuer, mais se pose la question de la granularité obtenue. Le développeur ayant à sa disposition plusieurs langages, il utilisera naturellement le flot de contrôle pour les parties de l'algorithme où le conditionnement et l'état sont prépondérant. Si le programme flot de contrôle est ensuite considéré comme un seul grain du point de vue de la distribution, comme c'est le cas de la chaîne Simulink/Scade/TTA, cela signifie que l'implantation se fait à partir d'une granularité dont est absente la notion de conditionnement. Le constat est le même si les programmes flot de données hétérogènes sont transformés en programmes flot de données homogènes avant la distribution. Cela correspond à une difficulté réelle de réaliser des distributions de programmes comportant des traitements conditionnés, car cela implique alors plusieurs cas d'exécution qu'il faut prévoir lors de la distribution, par exemple l'exécution de séquences exclusives entre elles et l'existence de plusieurs séquences de communications possibles.

Pourtant si la prise en compte du conditionnement dans la distribution rend plus complexe le problème de la distribution, elle permet d'implanter plus efficacement les algorithmes en utilisant au mieux l'architecture. Une FSM pourrait ainsi être distribuée sur l'architecture, au plus près des capteurs et des actionneurs correspondant à ses entrées et sorties. De même les différentes alternatives d'un algorithme pourront utiliser l'ensemble de l'architecture.

Pour cela il faut tout d'abord des traductions de programmes qui permettent de garder la granularité et le parallélisme potentiel des programmes flot de contrôle ou de tout autre langages

permettant le conditionnement. Ensuite, il faut un modèle flot de données permettant le conditionnement et surtout son implantation distribuée. Enfin, il faut des heuristiques permettant d'obtenir une implantation distribuée à partir d'un programme flot de données utilisant ce modèle.

Chapitre 2

Modèle flot de données permettant le conditionnement pour implantation distribuée

Nous allons d'abord montrer que les modèles flot de données existants qui permettent de décrire du conditionnement posent des problèmes lors d'une implantation distribuée. C'est pourquoi nous présentons ensuite notre propre modèle flot de données conditionné qui permet lui aussi d'exprimer le conditionnement mais permet en plus sa distribution efficace. Nous verrons dans les chapitres suivants comment des traductions permettent d'obtenir un programme flot de données conditionné unique à partir de programmes utilisant des langages permettant le conditionnement et comment le logiciel SynDEx permet d'obtenir une implantation distribuée à partir du programme flot de données conditionné obtenu.

2.1 Limites des modèles flot de données existants permettant le conditionnement

Nous avons vu dans l'état de l'art que la description du conditionnement dans le modèle flot de données nécessite d'étendre le modèle initial au modèle flot de données hétérogène, et d'employer des sommets spéciaux n'utilisant pas forcément toutes leurs entrées ou ne produisant pas forcément toutes leurs sorties. Ces sommets spéciaux sont déjà présents dans le modèle flot de données de Dennis sous la forme du couple de sommets *switch* et *merge* qui permet de décrire deux alternatives dans un même graphe. Ce modèle fût ensuite étendu par Buck dans son modèle d'Integer Data Flow[37] de manière à pouvoir décrire un nombre k d'alternatives. Ce modèle remplace les *switch* et *merge* de Dennis par des sommets *case* et *endcase*. La figure 2.1 page 38 montre un exemple de leur utilisation. Un sommet *case* possède deux ports d'entrée et k ports de sortie, $k \geq 2$. L'un des ports d'entrée est le port de contrôle et est étiqueté c. L'autre port d'entrée est étiqueté i. Les $k-1$ premiers ports de sorties sont étiquetés par des entiers, différents d'un port à l'autre. Le dernier port de sortie est étiqueté par *Def* pour "default". Lorsqu'il est exécuté, un sommet *case* consomme une

donnée sur chacun de ses ports d'entrée. Mais il ne produit une donnée que sur le port de sortie étiqueté par l'entier égal à la valeur consommée sur le port c, si un tel port de sortie existe, sinon il produit une donnée sur le port *Def*. Dans tous les cas, la valeur de la donnée produite est égale à celle de la donnée consommée sur le port d'entrée i.

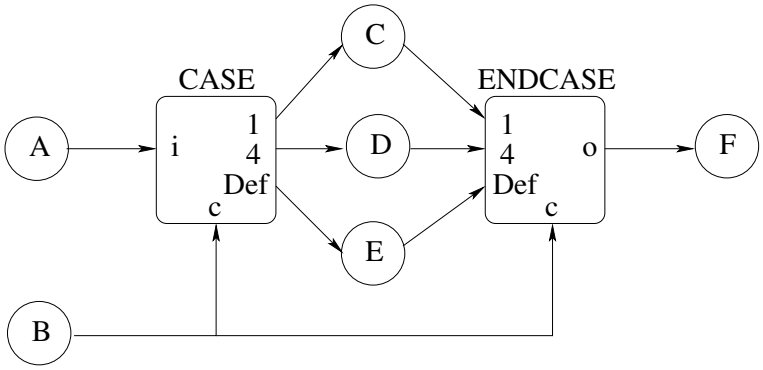

FIG. 2.1 – *Exemple de programme Integer Data Flow*

A chaque sommet *case* correspond un sommet *endcase*. Ce sommet *endcase* possède alors $k+1$ ports d'entrée et un port de sortie. L'un des ports d'entrée est le port de contrôle et est étiqueté c. Un autre port d'entrée est étiqueté *Def*. Les $k-1$ ports d'entrée restants sont étiquetés par des entiers, différents d'un port à l'autre mais correspondant à ceux étiquetant les ports de sortie du sommet *case* associé. Le port de sortie est étiqueté o. Lorsqu'il est exécuté, un sommet *endcase* consomme la donnée de son port de contrôle c ainsi que celle disponible sur le port d'entrée étiqueté par l'entier égal à la valeur consommée sur le port c, si un tel port d'entrée existe, sinon celle disponible sur le port *Def*. Dans tous les cas, une donnée est produite sur le port de sortie o et sa valeur correspond à celle de la donnée consommée sur le port d'entrée autre que le port de contrôle c.

Dans l'exemple donné par la figure 2.1 page 38, grâce au sommet "case", la valeur de la donnée produite par l'opération A est transmise à C si la valeur de la donnée produite par B est égale à 1, transmise à D si la valeur de la donnée produite par B est égale à 4 et transmise à E sinon. De même, grâce au sommet "endcase", le sommet F consomme une donnée dont la valeur est égale à celle produite par C si la valeur de la donnée produite par B est égale à 1, égale à celle produite par D si la valeur de la donnée produite par B est égale à 4 et égale à celle produite par E sinon. Ainsi, si les opérations A, B, F, *case* et *endcase* sont exécutées lors de chaque instant logique correspondant à une réaction du système, seule l'une des trois opérations C, D et E est alors exécutée.

Ce type de sommet permet de décrire du conditionnement, et couplé avec un ou plusieurs sommets "retard" permet de décrire l'équivalent d'une FSM. Néanmoins, contrairement au flot de données homogène, le flot de données hétérogène ne peut être directement exploité pour déduire une implantation distribuée. Nous allons ainsi montrer que le modèle d'Integer Data Flow ne représente pas toutes les dépendances de données nécessaires.

Considérons que les opérations qui composent le programme de la figure 2.1 page 38 sont distribuées sur des processeurs de telle manière que le C et F sont placées sur le processeur P_1

et le reste des opérations sur le processeur P_2. D'après le programme de la figure 2.1 page 38, le processeur P_1 ne nécessite pour l'exécution de C que la donnée provenant de l'opération *case*. La réception de cette donnée nécessite une communication de P_2 vers P_1 car le sommet *case* est placé sur P_2. En réalité, cette communication n'aura lieu que si la donnée produite par B vaut 1. En conséquence le processeur P_1 ne peut pas, sans connaissance de cette valeur, savoir si la communication va avoir lieu et s'il va devoir exécuter C. Cela signifie qu'il manque une dépendance de données entre l'opération B et l'opération C.

Ce modèle de flot de données incluant du conditionnement reste malgré tout la base du conditionnement dans le flot de données. Lorsqu'il est employé dans des outils qui permettent de faire de la distribution, comme c'est le cas dans Ptolemy, les opérations comprises entre les sommets *case* et *endcase* forment avec celles-ci un ensemble qui ne peut être séparé lors de la distribution, c'est-à-dire dont les opérations sont toutes placées sur le même processeur. Cela revient en fait à ne considérer pour la distribution que la partie homogène d'un programme flot de données hétérogène, ce qui résout le problème des dépendances de données manquantes. Ptolemy permettant la description hiérarchique (décrire un sommet par un graphe), il se peut qu'un programme de plusieurs milliers d'opérations soit contenu entre deux sommets *case* et *endcase*. Néanmoins ces opérations sont considérées comme une seule au moment de la distribution et le parallélisme n'est pas exploité. Il n'existe pas à notre connaissance de distribution de programme flot de données incluant du conditionnement.

Nous allons présenter notre modèle flot de données conditionné. Il peut être vu comme une version hiérarchique du modèle de Buck, ce qui permet d'utiliser moins d'arcs et de sommets. Mais notre principal apport consiste à rajouter automatiquement, une fois le graphe décrit, des sommets et des arcs afin de pouvoir distribuer le conditionnement du graphe sans problème de dépendances manquantes.

2.2 Modèle flot de données conditionné

Notre modèle flot de données conditionné consiste en la description d'un sous-graphe motif infiniment répété, chaque répétition correspondant à un instant logique des langages synchrones. C'est ce sous-graphe motif que par la suite nous désignerons par graphe. Les sommets du graphe sont des opérations qui doivent être exécutées avant que ne débute la prochaine répétition infinie du graphe. On parle ici de répétition infinie par rapport à répétition finie où dans le cas d'une boucle, une opération est répétée plusieurs fois. Nous préférons le terme répétition au terme itération car cette répétition n'est pas nécessairement temporelle (séquentielle) mais peut être aussi spatiale (parallèle). Chaque opération possède des ports d'entrée et/ou des ports de sortie et chaque dépendance de données relie un port de sortie à un ou plusieurs (diffusion) ports d'entrée. Il est à noter que si la diffusion est autorisée, plusieurs arcs ne peuvent relier différents ports de sortie au même port d'entrée (confusion). Un sommet spécifique appelé retard (comportement e'quivalent au sommet retard des langages flot de données) est nécessaire si une opération a besoin d'une donnée produite lors d'une précédente répétition infinie du graphe. On étend aussi le modèle flot de données de manière à rendre possible la description hiérarchique, une opération peut donc être décrite par un sous-graphe d'opérations.

Afin de pouvoir décrire le conditionnement, on introduit le principe des opérations conditionnantes et des opérations conditionnées, que nous avions présentées dans [54] et [55].

2.2.1 Opération conditionnante

Une opération conditionnante* est une opération qui est elle-même décrite par plusieurs sous-graphes. Un entier différent est associé à chacun de ces sous-graphes. Cette opération consomme une donnée appelée donnée de conditionnement* transmise par une dépendance de conditionnement*. Lors de chaque répétition infinie, seul le sous-graphe associé à l'entier égal à la valeur de la donnée de conditionnement est substitué à l'opération conditionnante. Les opérations constituant les sous-graphes d'une opération conditionnante sont appelées opérations conditionnées*. La condition* d'une opération conditionnée est le booléen correspondant au test "est-ce que la valeur de la donnée de conditionnement est égale à celle spécifiée pour le sous-graphe auquel j'appartiens ?". Une opération conditionnée n'est exécutée lors d'une répétition infinie du graphe que si sa condition est vraie. Grâce à la hiérarchie, une opération conditionnée peut aussi être décrite par un sous-graphe ou être à son tour une opération conditionnante.

Le graphe de la figure 2.2 page 40 montre un exemple de graphe flot de données conditionné. Il est constitué de 5 opérations non conditionnantes (A,B,D, E et F) et d'une opération conditionnante C. La donnée de conditionnement de C est la donnée produite par l'opération A :

- **si sortie(A)=1 :** l'exécution de C est équivalente à l'exécution de l'opération conditionnée C_{11} (sous-graphe en bleu);
- **si sortie(A)=2 :** l'exécution de C est équivalente à l'exécution de l'opération conditionnée C_{21} suivie de celle de l'opération conditionnée C_{22} (sous-graphe en vert).

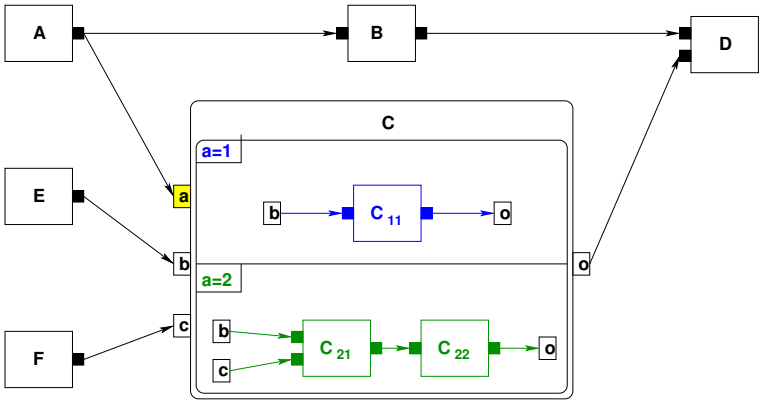

FIG. 2.2 – *Graphe flot de données conditionné*

En plus du port d'entrée correspondant à la donnée de conditionnement (en jaune sur la figure 2.2 page 40), une opération conditionnante possède des ports d'entrée et des ports de sortie qui peuvent être utilisés par les opérations des sous-graphes. Les ports d'entrée peuvent ainsi être connectés à n'importe quelles opérations des sous-graphes, sans limitation du nombre. Sur le graphe de la figure 2.2 page 40, le port b de l'opération conditionnante C est ainsi connecté à

l'opération C_{11} et à l'opération C_{21}. Ces deux opérations consommeront ainsi la donnée produite par le sommet E. Par contre les connexions entre les opérations conditionnées et les ports de sortie de l'opération conditionnante obéissent à une règle stricte.

En effet, le modèle flot de données impose qu'une donnée consommée doive d'abord être produite, sans quoi l'opération consommatrice ne peut commencer son exécution. En conséquence, une opération conditionnée ne peut être l'unique productrice d'une donnée consommée par une opération non conditionnée. Il suffirait lors d'une répétition infinie du graphe que la condition de l'opération conditionnée soit fausse pour que l'opération non conditionnée reste bloquée en attente de la donnée. Pour résoudre ce problème, on impose que chaque sortie d'une opération conditionnante soit dans chaque sous-graphe produite par une opération et une seule. Dit autrement, cela implique que dans chaque sous-graphe tous les ports de sortie de l'opération conditionnante soient connectés à exactement une opération. Dans l'exemple de la figure 2.2 page 40, les deux opérations conditionnées C_{11} et C_{22} sont productrices de la même donnée via le port de sortie o de C. Il ne s'agit pas de confusion de données car, d'une part ces deux opérations ne peuvent s'exécuter toutes les deux lors d'une même répétition infinie du graphe, et d'autre part nous verrons que la mise à plat de la hiérarchie en vue d'implantation distribuée substitue au port de sortie de l'opération conditionnante une opération possédant un port d'entrée pour chaque sous-graphe.

Il n'y a pas par contre de restrictions sur les ports d'entrée d'une opération conditionnante car une opération non conditionnée peut produire une donnée qui n'est consommée que par une opération conditionnée. Dans ce cas il se peut que la donnée soit produite mais pas consommée, ce qui n'est pas un problème. Dans l'exemple de la figure 2.2 page 40, la donnée produite par l'opération E est consommée par les deux opérations C_{11} et C_{21} via le port b de C alors que celle produite par F n'est consommée via le port c de C que dans un seul des deux cas, par C_{21}.

Deux opérations conditionnées du graphe sont dites en exclusion mutuelle* si leurs conditions respectives ne peuvent être vérifiées simultanément. On parle alors de conditions exclusives* . Cela signifie que lors de l'exécution temps réel, l'une d'entre elles seulement sera exécutée. Lors de la distribution et de l'ordonnancement cette propriété est intéressante car deux opérations en exclusion mutuelle peuvent alors être ordonnancées sur le même processeur en même temps. Sur le graphe de la figure 2.2 page 40, c'est le cas par exemple des opérations conditionnées C_{11} et C_{21}.

La figure 2.3 page 42 donne un exemple de distribution et d'ordonnancement sur une architecture distribuée composée de deux processeurs reliés par un bus. On y voit les opérations C_{11} et C_{21} être ordonnancées sur le même processeur durant le même intervalle de temps. À l'exécution, le processeur $Proc2$ reçoit via une communication la donnée produite par A sur le processeur $Proc1$. Cette donnée lui permet de savoir si c'est l'opération C_{11} ou C_{21} qui doit être exécutée. Si c'est la deuxième, le processeur doit ensuite attendre une communication venant de $Proc2$ pour connaître la donnée produite par F. Puisque les opérations C_{11} ou C_{21} ne sont jamais exécutées ensemble, on peut ainsi prévoir de les exécuter durant le même laps de temps.

2.2.2 Dépendance conditionnée et de conditionnement

Si l'une des opérations connectées par une dépendance est conditionnée, on parle de dépendance conditionnée*. Lors de l'implantation distribuée certaines dépendances impliquent des communications quand l'opération productrice et l'opération consommatrice sont implantées sur deux

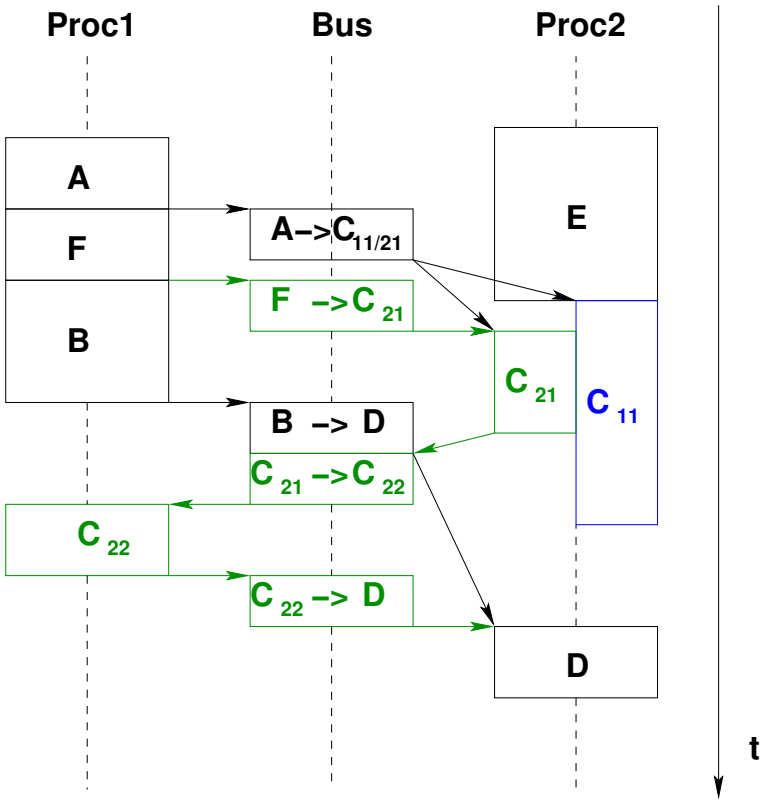

FIG. 2.3 – *Exemple de distribution et d'ordonnancement du graphe de la figure 2.2 page 40 sur une architecture distribuée*

processeurs distincts. Si la dépendance correspondante est conditionnée, on parle de communication conditionnée*. Sur notre exemple de distribution et d'ordonnancement de la figure 2.3 page 42, la dépendance de donnée $C_{21} \rightarrow C_{22}$ implique ainsi une communication de $Proc2$ vers $Proc1$. La condition d'une communication est la condition de l'opération consommatrice de la donnée communiquée. Par exemple, la condition d'une communication pour la dépendance $A \rightarrow B$ est la condition de B.

Une dépendance conditionnée est :

- soit entrante* (d'un prédécesseur du sous-graphe conditionné vers une des opérations du sous-graphe comme c'est le cas pour les dépendances $E \rightarrow C_{11}$, $E \rightarrow C_{21}$ (via le port b) et $F \rightarrow C_{21}$ (via le port c) sur la figure 2.2 page 40),
- soit interne* au sous-graphe conditionné (et relie donc deux opérations conditionnées ayant la même condition, comme c'est le cas pour la dépendance entre C_{21} et C_{22} sur la figure 2.2 page 40),
- soit sortante* (d'une des opérations du sous-graphe conditionné vers un des successeurs du sous-graphe comme c'est le cas pour les dépendances $C_{11} \rightarrow D$ et $C_{22} \rightarrow D$ (via le port o) sur la figure 2.2 page 40).

Enfin, on rappelle qu'une dépendance de conditionnement est une dépendance d'une opération produisant une donnée vers une opération pour laquelle cette donnée sert de donnée de conditionnement.

2.2.3 Ajout de dépendances et d'opérations pour implantation distribuée

La principale difficulté concernant le conditionnement est de s'assurer que, sur les processeurs effectuant une opération conditionnée, la ou les données de conditionnement de cette opération sont bien présentes avant d'effectuer cette opération (pour pouvoir tester si l'opération doit être exécutée ou non). De même l'émetteur et le ou les destinataires d'une communication conditionnée doivent tous connaître la ou les données de conditionnement. Pour résoudre ce problème, la mise à plat d'un graphe flot de données conditionné, c'est-à-dire la disparition de la hiérarchie par substitution des opérations par leurs sous-graphes, ajoute des opérations et des dépendances de données.

2.2.3.1 D'opération conditionnante à opération conditionnée

Dans le cas des opérations conditionnées, et afin d'assurer que le processeur exécutant une telle opération disposera de l'ensemble des données de conditionnement par lesquelles cette opération est conditionnée, on ajoute une dépendance de conditionnement pour chaque opération conditionnée d'une opération conditionnante. Cet ajout a lieu lors de la mise à plat quand les opérations conditionnantes sont remplacées par leurs sous-graphes. Il sera ainsi possible de tester la valeur de cette donnée pour savoir si l'opération doit être effectuée ou non. En effet lors de l'ordonnancement de cette opération conditionnée sur un processeur donné, les dépendances de conditionnement impliqueront des communications de la donnée de conditionnement vers ce processeur.

Par l'utilisation de plusieurs niveaux de hiérarchie, une opération peut-être conditionnée par plusieurs données de conditionnement. Ainsi, la condition d'une opération est une liste de conditions. Par exemple, une opération C appartenant à un des sous-graphes de l'opération conditionnée

B, elle-même appartenant à un des sous-graphes de l'opération conditionnée A, est conditionnée par la donnée de conditionnement de *A* et par celle de *B*. Cela se traduira donc, à la mise à plat, par l'ajout de deux dépendances de conditionnement sur l'opération C.

De plus cette liste de conditions est ordonnée. On parle ainsi d'imbrication de conditions et non d'ensemble de conditions. En effet à l'exécution les conditions doivent être testées dans l'ordre descendant de la hiérarchie du graphe. Si on reprend l'exemple de l'imbrication précédente A, B, C, lors de l'exécution on testera d'abord l'entrée de conditionnement de *A* (la condition pour laquelle *B* a lieu) puis l'entrée de conditionnement de *B* (la condition pour laquelle *C* a lieu). Ceci permet d'effectuer des regroupements, sous un même test, d'opérations conditionnées par la même condition. Il existe des techniques pour optimiser le regroupement de ces tests comme l'emploi de DDD [56] pour représenter l'imbrication des conditions. Cela permet d'optimiser le code généré après l'ordonnancement et la distribution. L'ajout des dépendances de conditionnement se fera donc de manière ordonnée et il existera un ordre sur les ports auxquels seront connectées ces dépendances.

Par la suite, ces dépendances de conditionnement que nous rajouterons seront reconnaissables car symbolisées par des arcs pointés (à la ligne faite de points).

2.2.3.2 Cas des dépendances conditionnées entrantes

Le prédécesseur d'une opération conditionnante ne connaît pas la donnée de conditionnement et ne peut donc pas savoir si la donnée qu'il produit va être utilisée et, si elle l'est, il ne peut savoir à quelle opération transmettre la donnée, puisque les opérations consommatrices peuvent appartenir à des sous-graphes différents de l'opération conditionnante.

Pour corriger cela on ajoute une opération *CondI**. Elle remplace la dépendance entrante, découpant celle-ci en deux étapes :

- elle consomme la donnée produite par le prédécesseur de l'opération conditionnante ainsi que la donnée de conditionnement de cette dernière. Tout comme ses prédécesseurs, un sommet *CondI* n'est pas conditionné, les dépendances dont il est la destination ne le sont donc pas ;
- parce que les opérations des sous-graphes sont conditionnées, la dépendance reliant le sommet *CondI* à l'une d'entre elles est conditionnée, ce qui n'est pas un problème étant donné que le sommet source (l'opération *CondI*) et le sommet destination ont connaissance de la donnée de conditionnement.

Afin de donner la sémantique précise des sommets *CondI*, nous allons adopter une méthode constructive en partant d'abord d'une sémantique naïve correspondant à un ajout d'autant de sommets *CondI* qu'il y a de dépendances entrantes. Puis nous tenterons de réduire le nombre de sommets *CondI* ajoutés en les factorisant, ce qui modifiera quelque peu à chaque fois leur comportement que nous donnerons sous la forme de pseudo-code.

Une première approche serait donc d'ajouter un tel sommet pour chaque dépendance entrante. Un sommet *CondI* posséderait alors deux port d'entrée, un pour la donnée de conditionnement et un pour la donnée produite par le prédécesseur de l'opération conditionnante, et un port de sortie pour se connecter à l'opération conditionnée du sous-graphe. Son comportement peut être représenté par le pseudo-code de l'algorithme 1 page 45.

Algorithme 1 Première version de l'algorithme correspondant à un sommet CondI

```
1: {La fonction CondI possède 4 paramètres : }
2: {- entrée_donnée_de_conditionnement}
3: {- entrée_donnée_pred}
4: {- sortie_donnée_op_conditionnée}
5: {- valeur_sous-graphe}
6: /*****************************/
7: consommation de entrée_donnée_de_conditionnement
8: consommation de entrée_donnée_pred
9: si entrée_donnée_de_conditionnement=valeur_sous_graphe alors
10:    sortie_donnée_op_conditionnée prend pour valeur entrée_donnée_pred
11:    production de sortie_donnée_op_conditionnée
12: fin si
```

Néanmoins, il est possible que plusieurs opérations d'un même sous-graphe consomment la même donnée venant du même prédécesseur. De plus, il se peut que ce prédécesseur produise plusieurs données consommées par des opérations d'un même sous-graphe. Dans ces deux cas, au lieu de multiplier les ajouts de sommets *CondI*, on factorise et son comportement est alors équivalent au pseudo-code de l'algorithme 2 page 46.

D'autre part, il est possible qu'une donnée produite par les prédécesseurs de l'opération conditionnante soit consommée par plusieurs opérations conditionnées appartenant à des sous-graphes différents de l'opération conditionnante. Sur l'exemple de la figure 2.2 page 40, c'est le cas des opérations C_{11} et C_{21} qui consomment la même donnée produite par E via le port b de l'opération C. Là encore, il est possible de factoriser le nombre de *CondI* à ajouter pour une opération conditionnante. Le nombre d'opérations *CondI* rajoutées pour une opération conditionnante est alors égal au nombre de prédécesseurs de cette opération ayant une dépendance conditionnée entrante sur cette opération [1].

Un sommet *CondI* possède donc autant de ports de sortie que le prédécesseur de l'opération conditionnante possède de ports de sortie dont les données sont consommées par des opérations des sous-graphes. Il comporte un port d'entrée de plus que de sortie car il doit consommer la donnée de conditionnement. Au final le pseudo-code correspondant au comportement d'une opération *CondI* est donné par l'algorithme 3 page 47.

La figure 2.4 page 48 montre le rajout de l'opération *CondI* pour les dépendances entrantes $E \rightarrow C_{11}$ et $E \rightarrow C_{21}$ (via le port b de C) de la figure 2.2 page 40. Les arcs en pointillés sont les dépendances conditionnées alors que les arcs pleins sont celles qui ne le sont pas. Enfin, nous

1. Attention, l'opération qui produit la donnée de conditionnement ne compte pas, bien qu'elle soit un prédécesseur. Cela reste vrai même si cette opération produit une autre donnée consommée par une opération d'un des sous-graphes ou si la donnée de conditionnement est consommée par une opération d'un des sous-graphes en tant que donnée et non donnée de conditionnement. En effet cette opération dispose dans tous les cas de la donnée de conditionnement.

Algorithme 2 Deuxième version de l'algorithme correspondant à un sommet CondI

```
1: {La fonction CondI possède 3 paramètres : }
2: {- entrée_donnée_de_conditionnement}
3: {- liste_entrées_sorties}
4: {- valeur_sous-graphe}
5: {liste_entrées_sorties contient des couples }
6: {(entrée_donnée_pred, sortie_donnée_op_conditionnée)}
7: /*********************************************/
8: consommation de entrée_donnée_de_conditionnement
9: pour chaque couple appartenant à liste_entrées_sorties faire
10:    consommation de entrée_donnée_pred
11: fin pour
12: si entrée_donnée_de_conditionnement=valeur_sous_graphe alors
13:    pour chaque couple appartenant à liste_entrées_sorties faire
14:       sortie_donnée_op_conditionnée prend pour valeur entrée_donnée_pred
15:       production de sortie_donnée_op_conditionnée
16:    fin pour
17: fin si
```

Algorithme 3 Version finale de l'algorithme correspondant à un sommet CondI

```
1: {La fonction CondI possède 3 paramètres : }
2: {- entrée_donnée_de_conditionnement}
3: {- liste_entrées_sorties_avec_valeurs}
4: {- liste_toutes_valeurs_sous-graphe}
5: {liste_entrées_sorties_avec_valeurs contient des triplets}
6: {(entrée_donnée_pred, sortie_donnée_op_conditionnée,liste_valeur_utile)}
7: {où liste_valeur_utile est la liste des valeurs de sous-graphes dans lesquels}
8: { la donnée d'entrée est effectivement consommée (utile)}
9: {liste_toutes_valeurs_sous-graphe contient la liste des valeurs de sous-graphes}
10: {dans lesquels au moins une des données de sortie est consommée}
11: /************************************************/
12: consommation de entrée_donnée_de_conditionnement
13: pour chaque triplet appartenant à liste_entrées_sorties_avec_valeurs faire
14:     consommation de entrée_donnée_pred
15: fin pour
16: pour chaque valeur appartenant à liste_toutes_valeurs_sous-graphe faire
17:     si entrée_donnée_de_conditionnement=valeur alors
18:         pour chaque triplet appartenant à liste_entrées_sorties_avec_valeurs faire
19:             si liste_valeur_utile contient valeur alors
20:                 sortie_donnée_op_conditionnée prend pour valeur entrée_donnée_pred
21:                 production de sortie_donnée_op_conditionnée
22:             fin si
23:         fin pour
24:     fin si
25: fin pour
```

rappelons que les arcs pointés sont les dépendances de conditionnement [2]. Les ports d'entrée des opérations conditionnées auxquels les dépendances de conditionnement sont connectées sont reconnaissables au fait qu'ils sont représentés par des carrés non pleins.

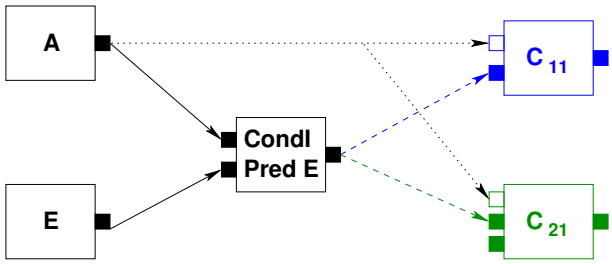

FIG. 2.4 – *Transformation des dépendances entrantes $E \rightarrow C_{11}$ et $E \rightarrow C_{21}$ de la figure 2.2 page 40*

2.2.3.3 Cas des dépendances conditionnées internes

Le cas des dépendances conditionnées internes n'entraîne aucun rajout d'opérations ni de dépendances. Les processeurs où sont ordonnancées l'opération productrice et l'opération consommatrice disposeront obligatoirement des données de conditionnement nécessaires grâce aux dépendances de conditionnement qui ont été ajoutées à toutes les opérations conditionnées lorsque l'opération conditionnante a été remplacée par ses sous-graphes.

Un exemple est donné dans la figure 2.5 page 48, sur lequel on peut voir les dépendances de conditionnement qui ont été rajoutées vers C_{21} et C_{22}.

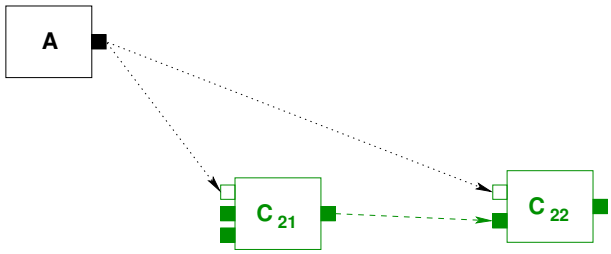

FIG. 2.5 – *Transformation de la dépendance interne $C_{21} \rightarrow C_{22}$ de la figure 2.2 page 40*

2.2.3.4 Cas des dépendances conditionnées sortantes

Un successeur d'une opération conditionnante n'est pas conditionné et à ce titre ne peut pas savoir quel sous-graphe d'opérations conditionnées s'est exécuté. D'autre part, il n'est pas possible

2. Un sommet *CondI* n'est pas conditionné et à ce titre la dépendance de données sur laquelle il consomme la donnée de conditionnement ne peut être considérée comme une dépendance de conditionnement. Elle apparaît donc sous la forme d'un arc plein.

que plusieurs dépendances, même si elles sont en fait exclusives, relient différentes données à un même port (confusion).

Pour chaque sortie d'opération conditionnante, on ajoute à la mise à plat une opération *CondO** qui se substitue au différentes dépendances sortantes utilisant le même port de sortie de l'opération conditionnante. Cette opération possède une entrée pour chacun des sous-graphes décrits plus une pour la donnée de conditionnement. En fonction de la valeur de cette dernière, l'opération *CondO* ne consomme que l'une de ses autres entrées, à savoir celle correspondant au sous-graphe associé à cette valeur. La donnée produite sur son unique port de sortie est alors égale à celle de la donnée consommée (produite par l'un des sous-graphe).

Les dépendances conditionnées sortantes sont remplacées par les dépendances de données suivantes :

- une dépendance de données conditionnée [3] pour chaque sous-graphe, reliant le port de sortie de l'opération conditionnée à l'un des ports d'entrée de l'opération *CondO*,
- une dépendance de données non conditionnée reliant le port de sortie de l'opération *CondO* au port d'entrée du successeur de l'opération conditionnante.

L'exécution d'un sommet *CondO* correspond au pseudo-code de l'algorithme 4 page 50.

La figure 2.6 page 49 montre l'ajout de l'opération *CondO* pour les dépendances sortantes $C_{11} \rightarrow D$ et $C_{22} \rightarrow D$. Enfin, la figure 2.7 page 50 montre le graphe mis à plat correspondant au graphe flot de données conditionné de la figure 2.2 page 40.

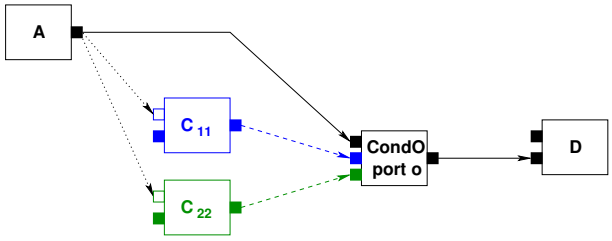

FIG. 2.6 – *Transformation des dépendances sortantes $C_{11} \rightarrow D$ et $C_{22} \rightarrow D$ utilisant la même sortie de C dans la figure 2.2 page 40*

2.2.3.5 Opérations conditionnantes conditionnées

Dans les explications précédentes, on a supposé qu'il n'y avait qu'un seul niveau de conditionnement et donc de hiérarchie, c'est-à-dire qu'une opération conditionnée ne pouvait être conditionnante. Cela nous permettait de parler d'opération et de dépendances non conditionnées. En fait il peut y avoir plusieurs niveaux de conditionnement et lorsqu'on utilise "non conditionné", il est sous-entendu "à ce niveau de hiérarchie". La figure 2.8 page 51 montre un exemple de graphe flot

3. Tout comme l'opération *CondI*, l'opération *CondO* consomme la donnée de conditionnement sans être conditionnée. Néanmoins elle a la particularité que les dépendances connectées à ses ports d'entrée soient conditionnées (hormis celle correspondant à la donnée de conditionnement) alors que habituellement cela impose que le destinataire le soit.

Algorithme 4 correspondant à un sommet CondO

```
1: {La fonction Cond0 possède 3 paramètres : }
2: {- entrée_donnée_de_conditionnement}
3: {- liste_entrée_donnée_et_valeur}
4: {- sortie_donnée}
5: {liste_entrées_donnée_et_valeur contient des couples}
6: {(entrée_donnée,valeur) qui pour chaque entrée indique}
7: {à quelle valeur de la donnée de conditionnement elle correspond}
8: /************************************************/
9: consommation de entrée_donnée_de_conditionnement
10: pour chaque couple appartenant à liste_entrée_donnée_et_valeur faire
11:     si entrée_donnée_de_conditionnement=valeur alors
12:         consommation de entrée_donnée
13:         sortie_donnée prend pour valeur entrée_donnée
14:     fin si
15: fin pour
16: production de sortie_donnée
```

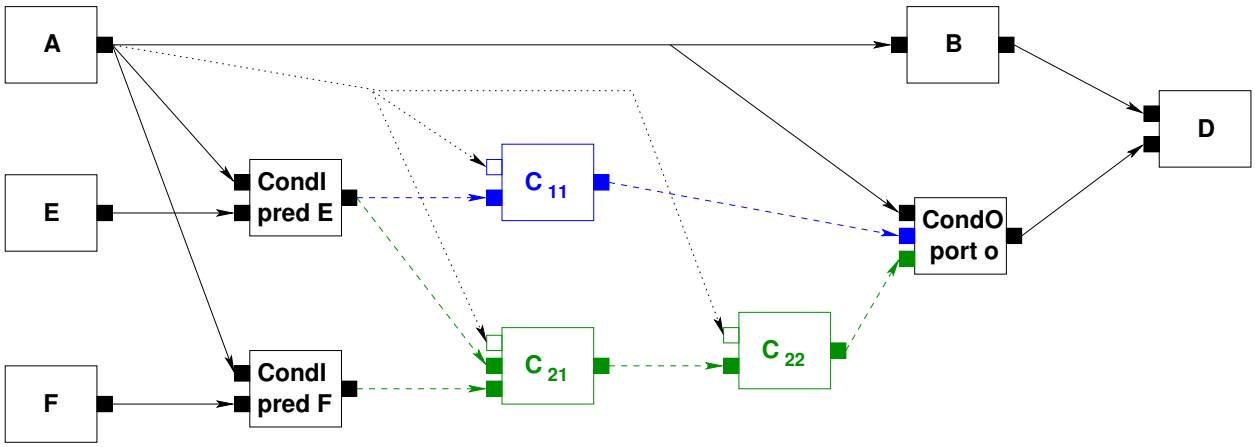

FIG. 2.7 – *Transformation du graphe flot de données conditionné de la figure 2.2 page 40*

de données conditionné avec deux niveaux de hiérarchie et la figure 2.9 page 52 montre le graphe obtenu après mise à plat.

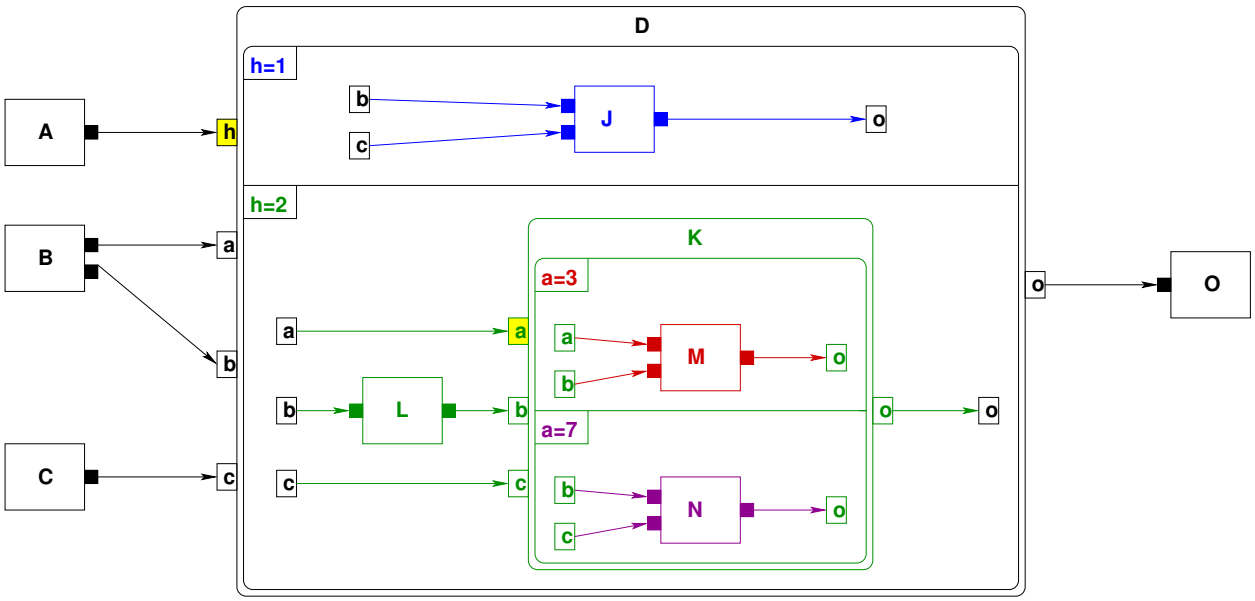

FIG. 2.8 – *Exemple de graphe flot de données conditionné à deux niveaux de conditionnement*

La mise à plat de l'opération *D* implique l'ajout de deux sommets *CondI* et d'un sommet *CondO*. En effet, bien qu'il existe trois ports d'entrée (*a*, *b* et *c*) connectés à des opérations des sous-graphes de *D*, ils n'ont que deux prédécesseurs distincts, les opérations *B* et *C*. Les deux sommets *CondI* ajoutés sont les sommets *CondI_Pred_B_D* (sommet *CondI* pour le prédécesseur *B* de l'opération conditionnante *D*) et *CondI_Pred_C_D*. Le sommet *CondO* rajouté est le sommet *CondO_Port_o_D* (sommet *CondO* pour le port *o* de l'opération conditionnante *D*). Un port d'entrée est ajouté aux opérations conditionnées *L* et *J* afin d'y connecter les dépendances de conditionnement issues de l'opération *A*.

La mise à plat de l'opération conditionnée et conditionnante *K* nécessite le rajout de deux sommets *CondI* et d'un sommet *CondO*. En effet, bien qu'il existe trois ports d'entrée connectés à des opérations des sous-graphes de *K*, l'un d'eux correspond à la donnée de conditionnement. Les deux sommets *CondI* rajoutés sont les sommets *CondI_Pred_L_K* et *CondI_Pred_C_K*. Le sommet *CondO* rajouté est le sommet *CondO_Port_o_K*. L'opération conditionnante *K* étant conditionnée, les trois sommets ajoutés héritent de sa condition et à ce titre sont munis d'un port supplémentaire pour y connecter la dépendance de conditionnement issue de l'opération *A*. Les opérations *M* et *N* sont munies de deux ports supplémentaires afin d'y connecter les dépendances de conditionnement issues respectivement de l'opération *A* (premier niveau de conditionnement) et de l'opération ajoutée *CondI_Pred_B_D* (deuxième niveau de conditionnement).

Sur la figure 2.9 page 52, les couleurs des opérations et des dépendances indiquent sous quelles conditions elles ont lieu et le trait des arcs (plein, pointillé ou pointé) permet de distinguer les dépendances conditionnées et de conditionnement.

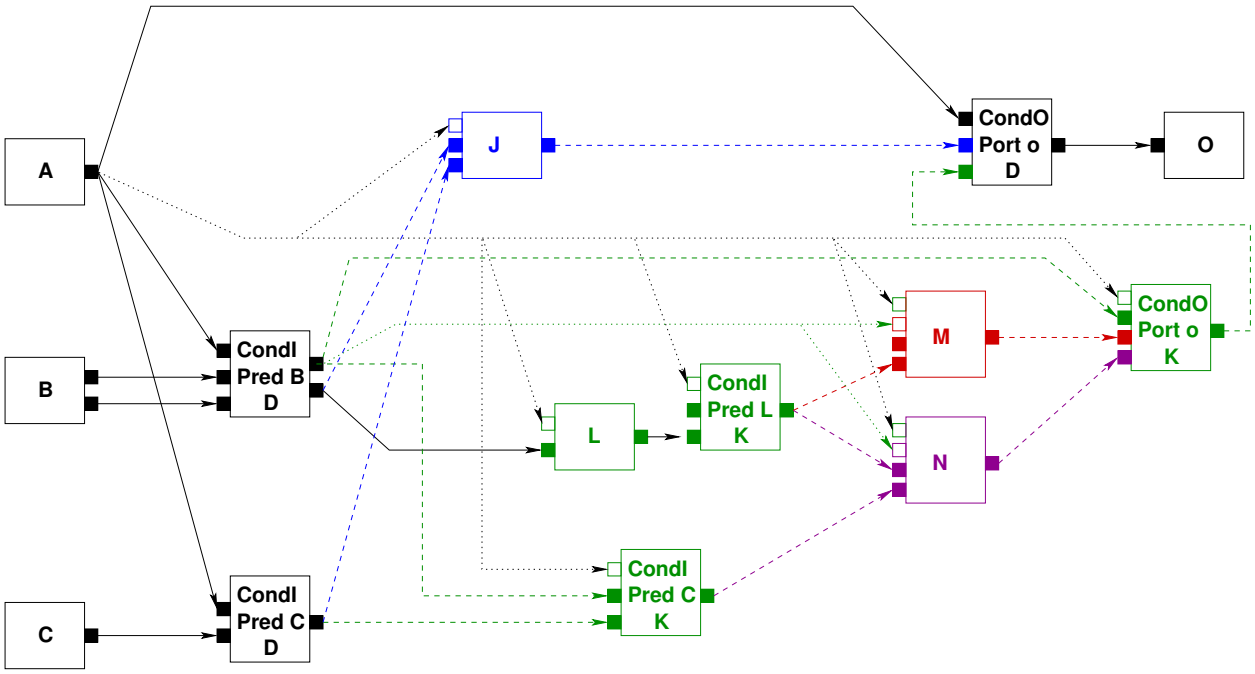

FIG. 2.9 – *Mise à plat du graphe flot de données conditionné de la figure 2.8 page 51*

2.3 SynDEx, langage utilisant le modèle flot de donnée conditionné

Ce modèle a permis d'étendre le langage SynDEx* [4] afin d'y décrire le conditionnement. En effet, il était jusqu'alors limité à la description de graphes flot de données homogène. Ainsi étendu, le langage SynDEx est, d'un point de vue graphique, identique au modèle flot de données conditionné présenté. Ainsi lorsqu'on parlera du langage SynDEx, le lecteur pourra se référer à ce chapitre. Ce langage a pour but de permettre la description de fonctionnalités à implanter. Le programme est ensuite transformé, le graphe flot de données conditionné correspondant étant mis à plat selon la méthode énoncée précédemment (voir 2.2.3). Ce langage est donc adapté lorsqu'une implantation distribuée du programme est visée. Nous présentons plus loin le logiciel SynDEx qui utilise le langage du même nom pour représenter l'algorithme d'un système et permet ensuite de représenter une architecture distribuée et d'obtenir automatiquement l'implantation de l'algorithme sur l'architecture.

4. www.syndex.org/v6/grammar/grammar.html

Chapitre 3

Traductions de langages permettant le conditionnement en langage SynDEx

3.1 Nécessité de traductions

Notre modèle flot de données conditionné a comme avantage principal de permettre l'implantation distribuée du conditionnement. Cependant, nous ne prétendons pas qu'il soit le mieux adapté pour la description du conditionnement et la simulation. C'est pourquoi nous proposons de traduire automatiquement des langages qui y sont mieux adaptés, en langage SynDEx qui est basé sur notre modèle flot de données conditionné.

Nous décrivons ici les méthodes de traduction concernant deux langages. La première concerne la traduction du langage SyncCharts qui est représentatif des langages de type FSM. La deuxième concerne la traduction du langage Scicos qui utilise le modèle schéma-bloc de Simulink en y introduisant le conditionnement. Le modèle schéma-bloc ressemble au flot de données mais la règle d'activation n'y est pas respectée : la destination d'un arc peut s'exécuter même si la donnée correspondante n'a pas été produite. La donnée consommée est alors la dernière à avoir été produite, on dit qu'il y a rémanence* de la donnée. Ce modèle dérivé du modèle flot de données étant très utilisé pour modéliser et simuler les systèmes (lois de commande notamment), il nous a semblé intéressant de montrer la traduction d'un des langages utilisant ce modèle.

3.2 Traduction SyncCharts/SynDEx

3.2.1 Des FSM non hiérarchiques à SyncCharts

Les premiers langages flot de contrôle de type FSM étaient non hiérarchiques. Or, lors de la conception d'une application, on constate que le nombre d'états de la FSM croît exponentiellement par rapport à la taille de l'application. Avec le succès croissant des méthodes de développement bottom-up et top-down (respectivement montante et descendante) dans les années 80, il devint nécessaire de proposer la description hiérarchique, ce que fit David Harel en proposant le langage Statecharts en 1984 [29][57]. En plus de la hiérarchie, grâce à laquelle un état peut être décrit par

une FSM, Statecharts permet également la composition de FSM concurrentes pouvant alors évoluer en parallèle. Ces FSM hiérarchiques et concurrentes utilisent le principe de diffusion immédiate : chaque changement d'un état dans une des FSM ou la modification d'une donnée par l'un d'eux est immédiatement perçu par l'ensemble des FSM concurrentes ou hiérarchiques. Dans Statecharts, les FSM gagnent en expressivité et modularité comme le montre l'exemple de la figure 3.2.1 page 54. Cette figure présente à gauche un FSM non hiérarchique à gauche et son équivalent Statecharts à droite : le nombre d'état y est réduit par la composition parallèle et les transitions vers l'état z sont factorisées par la hiérarchie.

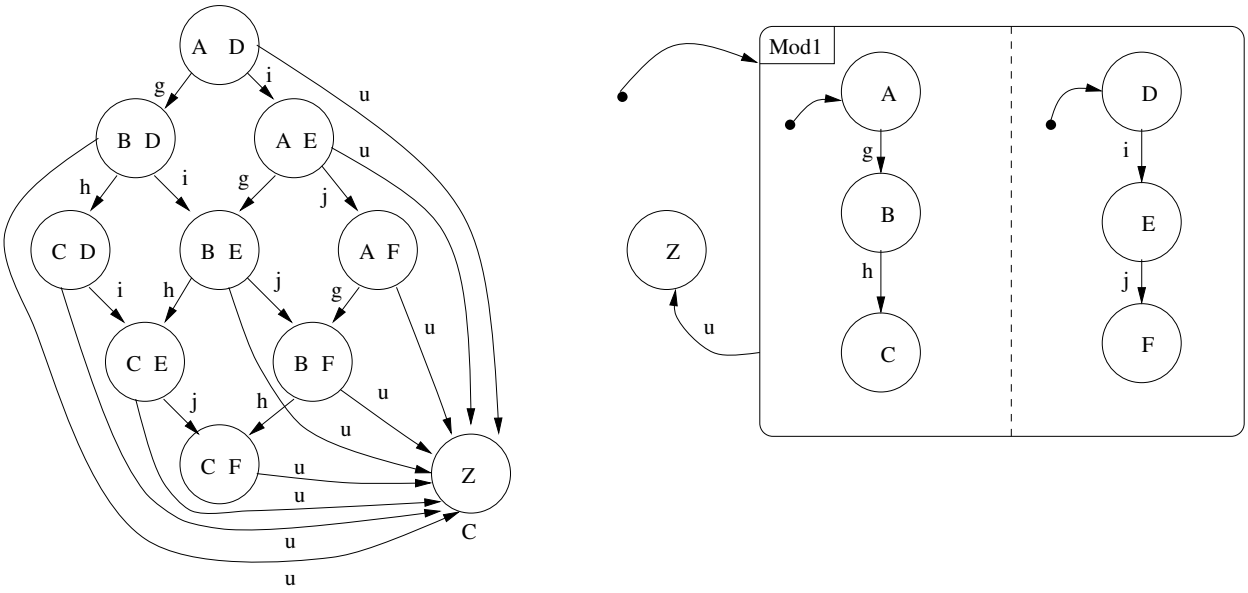

FIG. 3.1 – *FSM non hiérarchique et équivalent Statecharts*

Précurseur en matière de formalisme flot de contrôle visuellement explicite, le langage Statecharts a connu et connaît encore un grand succès. Il en existe une version UML et le langage Stateflow inclus dans Simulink s'en inspire également. Il en existe beaucoup d'autres similaires car si Statecharts a introduit de nouveaux concepts (hiérarchie, composition) sa sémantique initiale comportait quelques lacunes. Celles-ci sont décrites dans [58] où l'auteur référence également les différents langages dérivés de Statecharts en donnant les lacunes comblées par chacun d'entre eux. A titre d'exemple, une des lacunes principales est le manque de déterminisme. Il est en effet possible dans une FSM Statecharts qu'un état puisse être quitté par deux arcs différents dont les conditions ne sont pas exclusives. Si les conditions sont vraies lors de la même réaction du système on ne peut alors pas savoir quel arc sera franchi et donc quel sera l'état suivant. Une autre lacune est de ne pas garantir la causalité en ne définissant pas clairement combien d'arcs peuvent être franchis lors d'une réaction du système, et sous quelles conditions le franchissement d'un arc peut entraîner le franchissement d'un autre.

Le formalisme SyncCharts (acronyme de *Synchronous Charts*), lui aussi inspiré de Statecharts, a été introduit en 1995 par Charles André [59]. Ce formalisme peut être vu comme une forme

graphique du langage synchrone Esterel, ce qui implique une sémantique plus stricte que celle de Statecharts. Il est par exemple obligatoire de préciser des priorités entre les arcs si plusieurs permettent de quitter le même état. De plus, la possibilité de générer du code Esterel permet ensuite de détecter les erreurs de causalité grâce au compilateur Esterel. La causalité constructive propre à SyncCharts et Esterel sera abordée plus loin (voir 3.2.3.4 page 66). On utilise le terme SyncCharts pour parler du langage et on utilise SyncChart pour désigner un programme utilisant ce langage.

3.2.2 Le langage SyncCharts

La figure 3.2 page 56 résume les éléments graphiques qui composent les SyncCharts* . Les éléments de base sont :

- une étoile* (*star*, figure 3.2 page 56, partie B) est constituée d'un état et des différents arcs permettant de quitter cet état. Les arcs issus d'un état constituent les "rayons" de l'étoile ;
- une constellation* (figure 3.2 page 56, partie C) est un ensemble d'étoiles tel qu'une et une seule étoile est active à l'intérieur d'une constellation ;
- un macro-état* (*firmament*, figure 3.2 page 56, partie A) est constitué d'une ou plusieurs constellations en parallèle.

On a donc un seul état courant (actif) dans une constellation mais il peut y en avoir plusieurs dans un macro-état s'il est composé de constellations en parallèle. Nous emploierons fréquemment par la suite les termes : étoile, constellation et macro-état. Dans chaque constellation, l'état initial est l'état destination de l'arc dont la source n'est pas un état mais un rond noir. Lorsque le macro-état contenant cette constellation devient actif, c'est cet état qui est actif dans la constellation. Les états actifs peuvent changer lors de chaque instant logique lorsque les constellations sont évaluées.

Les entrées et les sorties d'un programme SyncCharts sont appelées signaux, reprenant en cela la terminologie Esterel. A ceux-ci peuvent s'ajouter des signaux locaux utilisés par les différentes constellations d'un macro-état pour se synchroniser. Lors de chaque instant logique, un signal peut être soit présent, c'est-à-dire être émis, soit absent. Si on note I l'ensemble des signaux d'entrée d'un graphe SyncCharts et O l'ensemble des signaux de sortie, lors du $k^{ème}$ instant logique seul un sous-ensemble des signaux de I peut être présent et seul un sous-ensemble des signaux de O peut être émis, et donc présent lors de cet instant logique [1].

Le but de l'évaluation des constellations d'un programme est de décider, à partir de l'ensemble des signaux d'entrée présents, quels sont les signaux de sortie qui le seront. Cela impose que l'ensemble des signaux d'entrée présents soit connu et fixe au cours de l'évaluation, et donc que $I \cap O = \emptyset$. En plus de leur présence et de leur absence les signaux peuvent être valués. L'émission d'un signal (présence) ou sa modification (signal valué) est instantanément perçue dans l'ensemble du graphe.

Les transitions qui étiquettent les arcs sont de la forme :

$$trigger[condition]/effect$$

1. L'hypothèse synchrone implique que la présence des signaux d'entrée ait lieu au même instant logique que l'émission des signaux de sortie, les causes et les effets se superposant.

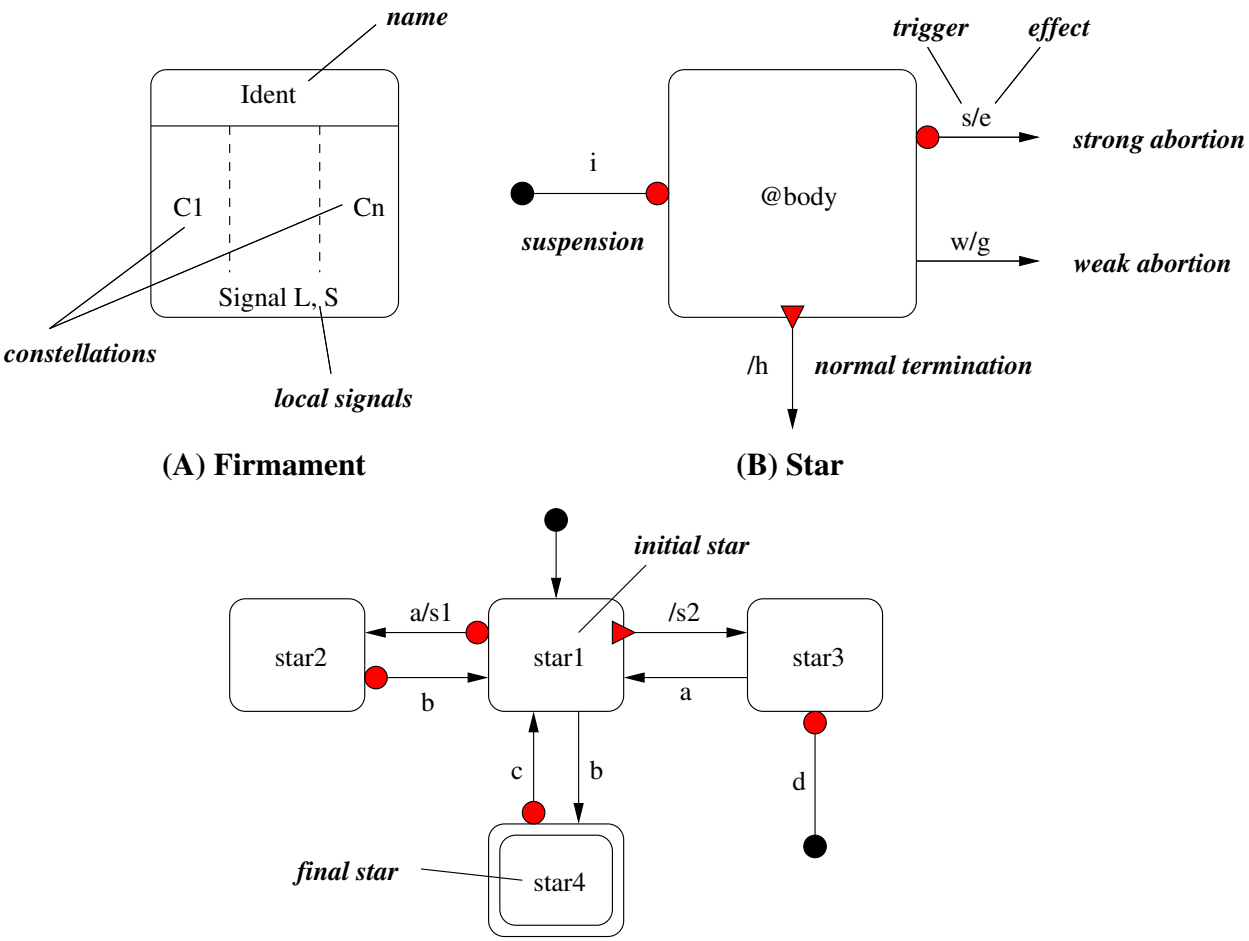

FIG. 3.2 – *Éléments de Syntaxe SyncCharts*

où :

- *trigger* est une expression booléenne utilisant les opérateurs booléens AND, OR, NOT et des parenthèses, portant sur la présence des signaux,
- *condition* est un prédicat pouvant faire intervenir la valeur des signaux,
- *effect* est une liste de signaux, éventuellement valués, à émettre lors du franchissement de l'arc.

Aucun des trois champs n'est obligatoire. Il est par ailleurs possible d'allouer un champ *effect* à un état. Les signaux concernés sont alors émis et/ou modifiés tant que l'état en question est l'état courant.

Il existe dans SyncCharts trois types d'arcs différents permettant de passer d'un état à un autre :

- préemption forte* (*strong abortion*, figure 3.2 page 56, partie B). Un arc de ce type est caractérisé par un rond rouge à son origine. L'état quitté est préempté immédiatement, les effets

qui lui sont attachés ne sont pas produits et, s'il s'agit d'un macro-état, les constellations contenues ne sont pas évaluées ;

– préemption faible* (*weak abortion*, figure 3.2 page 56, partie B). Un arc de ce type est un arc sans décoration. L'état quitté est préempté mais les effets qui lui sont attachés sont produits et, si l'état quitté est un macro-état, les constellations contenues sont évaluées ;
– terminaison normale* (*normal termination*, figure 3.2 page 56, partie B). Un arc de terminaison normale a pour origine un triangle rouge et ne possède pas de champ *trigger* ni de champ *condition*. Sa source est forcément un macro-état dans lequel chaque constellation possède un état final reconnaissable à son bord double (*final star*, figure 3.2 page 56, partie C). Si le macro-état est l'état courant et que, dans chaque constellation qu'il contient, l'état courant est l'état final, l'arc de terminaison normale peut être franchi.

En plus de ces arcs, il en existe un de particulier car ne permet pas de quitter un état, c'est l'arc de suspension*. Il ne possède pas d'état source et lorsque l'expression de son champ *trigger* est vraie, le macro-état destination est suspendu. Cela signifie que les états courants des constellations qu'il contient ne peuvent changer. Par contre les effets attachés à ces états ne sont pas suspendus. Lorsqu'il est suspendu, un macro-état peut être malgré tout préempté. Graphiquement, ces arcs ont pour origine un rond noir et pour terminaison un rond rouge (*suspension*, figure 3.2 page 56, partie B).

Il existe des priorités entre les arcs : un arc de préemption forte est plus prioritaire qu'un arc de préemption faible qui est plus prioritaire qu'un arc de terminaison normale. Si plusieurs arcs du même type permettent de quitter un même état, il faut préciser des priorités entre eux.

Chacun des arcs permettant de quitter un état pour un autre est instantané* si un caractère # précède son champ *trigger*. En absence de tels arcs, un seul arc peut être franchi par constellation au cours d'un instant logique. Mais si un arc est franchi et que l'état destination est la source d'un ou de plusieurs arcs instantanés, l'un de ces arcs peut être franchi au cours du même instant logique.

Afin d'illustrer ce formalisme, nous empruntons à C. André le fameux exemple ABRO [60]. Il s'agit de la description des fonctionnalités d'un système à trois entrées A B et R et une sortie O. Son comportement est le suivant :

– le système attend en parallèle l'occurrence des signaux A et B,
– quand les deux ont eu lieu, il y a émission du signal O,
– durant l'attente, il peut y avoir préemption par présence du signal R,
– chaque occurrence de R ré-initialise le système.

La figure 3.3 page 58 représente la description de l'algorithme par un programme SyncCharts. On peut y observer les différentes formes d'arcs (préemption forte, terminaison normale) et d'états (états, macro-états, états finaux).

3.2.3 Principes de la traduction

Nous avons choisi SyncCharts à cause de sa sémantique déterministe et plus stricte que celle de Statecharts. L'idée est néanmoins que les principes énoncés ici soient transposables à n'importe quel langage "à la Statecharts". C'est pourquoi nous commencerons par énoncer les principes de

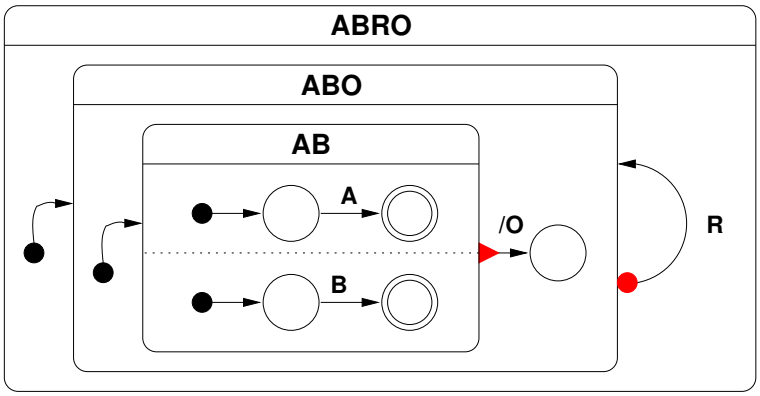

FIG. 3.3 – *Exemple de programme SyncCharts : ABRO*

traduction des entités que l'on retrouve dans tous ces langages (parfois sous des noms différents) à savoir la FSM non hiérarchique, la composition parallèle, la hiérarchie et la synchronisation par signaux locaux.

Afin de faciliter la compréhension, on se restreindra dans un premier temps à des signaux non valués avant de décrire à la fin comment ils peuvent être introduits. Nous verrons alors comment l'émission ou la modification de signaux peut être remplacée par n'importe quelle fonction de calcul. Pour l'instant, les étiquettes des arcs seront donc restreintes à *trigger/effect*.

De même, dans un premier temps, nous considérons que les arcs sont tous des arcs de préemption forte, avant d'introduire ensuite la traduction des autres types d'arcs.

3.2.3.1 Traduction d'une constellation

On appellera état courant* le nouvel état calculé lors d'un instant logique et état précédent* l'état calculé lors de l'instant logique précédent. Dans une constellation il n'y a qu'un état actif à la fois. De l'état précédent lors d'un instant logique dépend les arcs qui vont pouvoir être franchis et donc ceux à évaluer dans l'ordre de leur priorité.

S'il existe N_e états dans la constellation, on peut représenter l'état précédent de la constellation par un entier k tel que $1 \leq k \leq N_e$. Afin que l'état précédent soit disponible pour calculer l'état courant lors d'un instant logique, il est obligatoire en flot de données d'utiliser une opération *retard*. Cette opération *retard* consomme l'entier représentant l'état courant après avoir produit l'entier représentant l'état précédent.

La traduction d'une constellation en un graphe SynDEx correspond, en ce qui concerne le plus haut niveau de la hiérarchie, au graphe de la figure 3.4 page 59. L'état précédent est disponible grâce à l'opération retard $\$Etat$. L'opération conditionnante *CalculEtatCourant* consomme l'état précédent comme donnée de conditionnement et produit l'état courant en sortie. Cette opération possède donc N_e sous-graphes.

S'il existe N_i signaux d'entrée à ce SyncChart[2] et N_o signaux de sortie, l'opération condi-

2. Rappel : SyncCharts avec un "s" désigne le langage alors qu'un SyncChart, sans "s" désigne un programme.

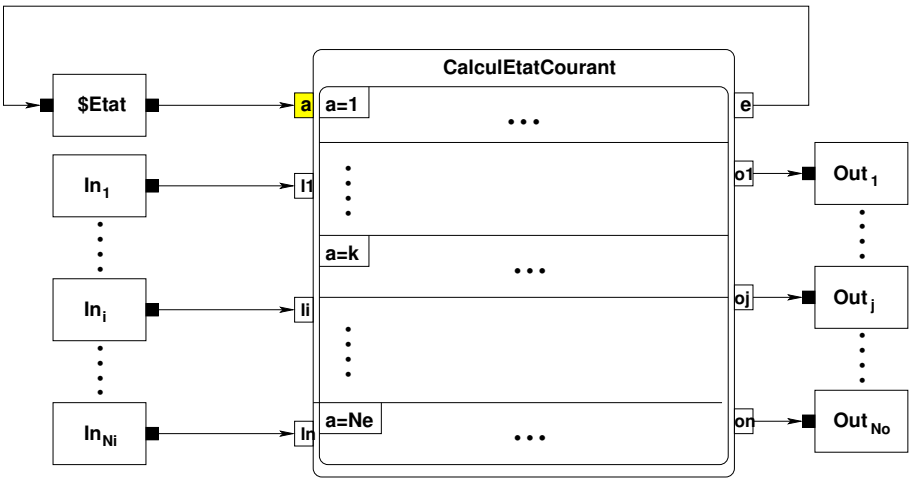

FIG. 3.4 – *Premier niveau hiérarchique de la traduction d'une constellation*

tionnante possède $(N_i + 1)$ ports d'entrée et $(N_o + 1)$ ports de sortie. Sur la figure 3.4 page 59, les opérations In_x et Out_x représentent respectivement les capteurs et les actionneurs permettant respectivement d'acquérir les signaux d'entrée et de retourner les signaux de sortie.

Intéressons nous à présent aux sous-graphes de l'opération conditionnante *CalculEtatCourant*. Chacun correspond à une étoile de la constellation qui est, nous le rappelons, un état et les arcs qui permettent de quitter cet état. Chaque sous-graphe ne nécessite donc qu'un sous-ensemble des signaux d'entrée, ceux utilisés par les champs *trigger* des arcs de l'étoile correspondante et n'utilise ainsi qu'une partie des ports d'entrée de l'opération conditionnante. Les arcs étant munis de priorités, on évalue leur franchissement dans l'ordre jusqu'au premier franchissable ou, le cas échéant, jusqu'à les avoir tous évalués.

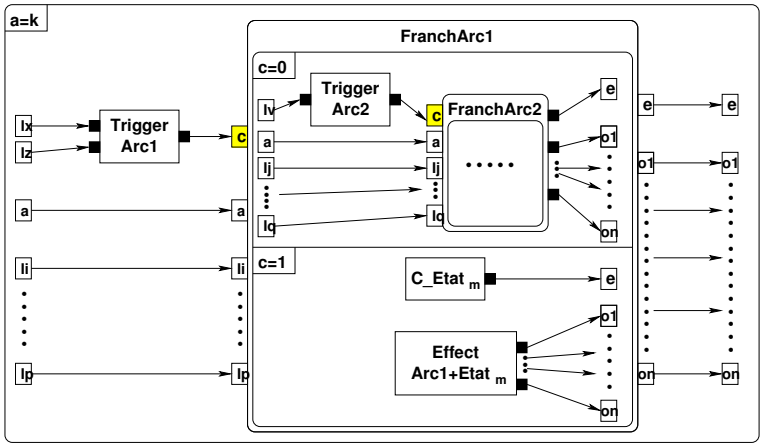

FIG. 3.5 – *Structure des sous-graphes appartenant à CalculEtatCourant*

La figure 3.5 page 59 montre le sous-graphe de *CalculEtatCourant* correspondant à l'état pré-

cédent représenté par l'entier k. L'opération *TriggerArc1* consomme en entrée les signaux utilisés dans l'expression booléenne du champ *trigger* de l'arc le plus prioritaire partant de l'état k. Elle produit l'entier 1 si l'expression est vraie, 0 sinon. Par la suite, toutes les opérations *TriggerArcX* auront la même fonctionnalité.

La donnée produite par l'opération *TriggerArc1* permet de conditionner l'opération $FranchArc1$ chargée de calculer l'état courant et les signaux de sortie. Cette opération, comme toutes les opérations $FranchArcX$, comporte deux sous-graphes. Si l'arc est franchissable, la valeur de la donnée de conditionnement produite par *TriggerArc1* vaut 1 et le sous-graphe comporte alors deux opérations :

- C_Etat_m produit une constante dont la valeur m correspond à l'état destination de l'arc franchi. Cette donnée représente l'état courant et est consommée via le port e de $FranchArc1$ et le port e de $CalculEtatCourant$ par l'opération $\$Etat$, afin de devenir l'état précédent lors du prochain instant logique. Toutes les opérations C_Etat_X possèdent la même fonctionnalité ;
- $EffectArc1Etat_m$ produit les valeurs booléennes correspondant aux signaux de sortie (émission est codée 1, non-émission correspond à 0). Les signaux émis sont ceux correspondant au champs $effect$ de l'arc franchi ($Arc1$) et de l'état destination ($Etat_m$). Parce qu'une opération conditionnante nécessite la production de toutes ses sorties, cette opération produit pour tous les autres signaux une donnée correspondant à la non-émission (0). Toutes les opérations $EffectArcXEtat_Y$ possèdent la même fonctionnalité.

Si l'arc n'est pas franchissable, le sous-graphe correspondant (pour $c = 0$ sur la figure 3.5 page 59) peut prendre deux formes suivant qu'il existe ou non un autre arc moins prioritaire, permettant de quitter l'état k. Sur la figure 3.5 page 59, c'est le cas où il existe un autre arc qui est représenté. Le sous-graphe prend alors une forme en tout point semblable à celui du sous-graphe de $CalculEtatCourant$. Il s'agit en effet d'évaluer le franchissement d'un nouvel arc, en utilisant une opération *TriggerArc2* et une opération conditionnante $FranchArc2$. Les évaluations successives du franchissement des différents arcs permettant de quitter un état k consistent donc en une imbrication d'opérations conditionnantes de type $FranchArcX$.

L'opération $FranchArcX$ correspondant à l'arc le moins prioritaire possède un sous-graphe différent pour le cas où cet arc n'est pas franchissable ($c = 0$). En effet cela signifie que l'état précédent k va devenir (rester) l'état courant. Dans ce cas, l'opération $FranchArcX$ correspond à la figure 3.6 page 62. Il n'y a pas de différence pour le cas $c = 1$ où l'arc est franchi, mais pour le cas $c = 0$ où l'état reste inchangé :

- l'état précédent devient l'état courant grâce à la dépendance de donnée connectant le port d'entrée a au port de sortie e. En effet via la hiérarchie du conditionnement, ces ports correspondent à ceux connectés à l'opération $\$Etat$ (voir figures 3.4 page 59 et 3.5 page 59) ;
- les signaux de sortie sont produits par une opération $EffectEtat_k$, qui émet (produit une donnée de valeur 1) les signaux correspondant au champ $effect$ de l'état k et produit 0 pour la non-émission des autres.

Cette description de la constellation sous la forme d'un graphe SynDEx correspond donc à l'imbrication de conditions correspondant au pseudo-code de l'algorithme 5 page 61.

Il peut exister, dans la constellation, des états qu'il n'est pas possible de quitter car ils ne sont la source d'aucun arc. Un tel état est appelé état-puits. Le sous-graphe de l'opération $CalculEtatCourant$

Algorithme 5 d'imbrications de conditions

```
1: si l'état précédent vaut 1 alors
2:     {Cas 1}
3: fin si
4: ...
5: {Tests des cas 2 à k − 1}
6: ...
7: si l'état précédent vaut k alors
8:     si l'arc le plus prioritaire est franchissable alors
9:         état courant = destination de cet arc
10:        émettre les signaux des effets de l'arc et de l'état courant
11:    sinon
12:        si l'arc moins prioritaire est franchissable alors
13:            état courant = destination de cet arc
14:            émettre les signaux des effets de l'arc et de l'état courant
15:        sinon
16:            {pas d'arc moins prioritaire}
17:            état courant = état précédent
18:            émettre les signaux des effets de l'état précédent
19:        fin si
20:    fin si
21: fin si
22: ...
23: {Tests des cas k + 1 et supérieurs}
24: ...
```

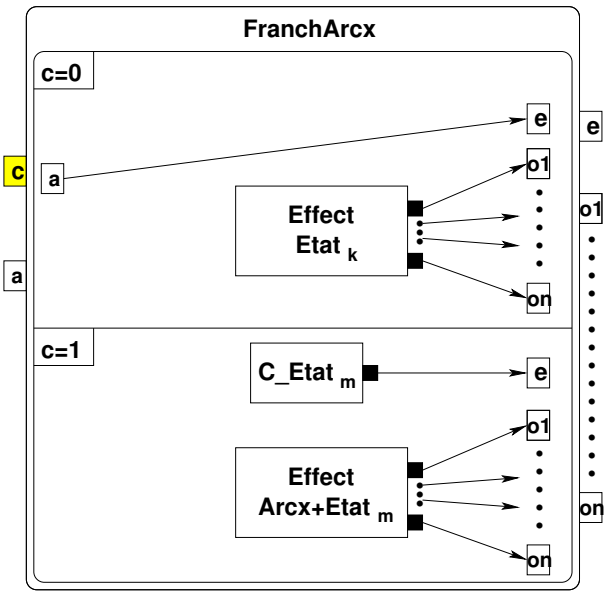

FIG. 3.6 – *Structure d'une opération FranchArcX quand l'arc x est l'arc le moins prioritaire*

correspondant à un état-puits, illustré par la figure 3.7 page 62, ne contient pas d'opération *FranchArcX* et ressemble beaucoup au sous-graphe d'une opération *FranchArcX* correspondant au non-franchissement de l'arc le moins prioritaire (voir figure 3.6 page 62).

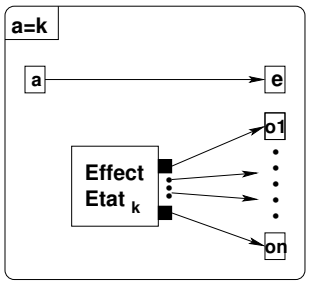

FIG. 3.7 – *Structure d'un sous-graphe appartenant à CalculEtatCourant et correspondant à un état-puits*

Pour ce qui est de l'état initial, il faut s'assurer que lors du premier instant logique, ce sera le bon sous-graphe de *CalculEtatCourant* qui sera exécuté. Pour cela, il suffit d'assigner l'entier codant l'état initial comme valeur initiale d'une opération $*Etat*. En effet cette opération nécessite une valeur initiale pour produire une donnée lors de la première répétition infinie du graphe SynDEx.

Enfin, en ce qui concerne les signaux d'entrée, l'imbrication des opérations conditionnantes implique qu'une opération conditionnante de niveau hiérarchique *n* (n=0 étant le plus haut niveau, celui représenté par la figure 3.4 page 59) doit posséder les ports d'entrée correspondant aux si-

gnaux d'entrée nécessaires pour les opérations de niveaux hiérarchiques inférieurs c'est-à-dire pour tous les niveaux m tel que $m > n$. Ainsi une opération *FranchArcX* possède des ports d'entrée correspondant aux signaux d'entrée utilisés par les champs *triggers* des arcs moins prioritaires.

3.2.3.2 Traduction de la composition parallèle

La traduction de la composition parallèle ne pose pas de problème particulier tant qu'il n'y pas de signaux locaux échangés entre les différentes constellations de la composition parallèle, problème que nous traiterons plus loin (au 3.2.3.4 page 66). Les constellations sont traduites séparément et les graphes SynDEx en résultant sont mis en parallèle au même niveau de hiérarchie.

La figure 3.8 page 63 montre à gauche un exemple de composition parallèle dans SyncCharts et à droite la traduction en graphe SynDEx avec la représentation du plus haut niveau de hiérarchie. On remarquera que sur le graphe SyncCharts, il existe un état pouvant être quitté par deux arcs différents. L'entier à la source des deux arcs correspondants indique la priorité ($i < j$ implique que i est plus prioritaire que j).

Les deux constellations pouvant partager les signaux d'entrée et de sortie, il est nécessaire pour les signaux de sortie de faire un consensus sur l'émission ou la non-émission des signaux, c'est-à-dire considérer présent un signal émis par au moins une des constellations et le considère absent sinon. C'est le cas sur la figure 3.8 page 63 avec le signal de sortie D. Puisque notre représentation de l'émission ou la non-émission des signaux est booléenne, il suffit pour les signaux de sortie partagés d'ajouter une opération OR correspondant à l'opérateur booléen du même nom.

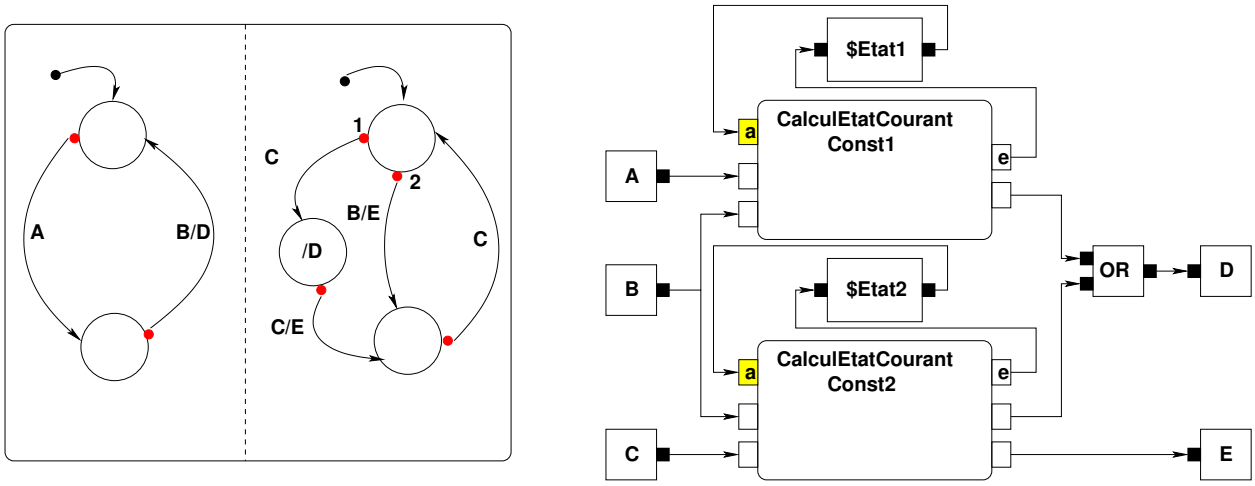

FIG. 3.8 – *Exemple de traduction de composition parallèle*

3.2.3.3 Traduction d'un macro-état (hiérarchie)

L'idée est de garder notre représentation d'une constellation, c'est-à-dire un couple (opération retard $\$Etat$, opération conditionnante *CalculEtatCourant*), et de considérer la ou les constella-

tions composant un macro-état sur le même plan que l'émission des signaux de sortie appartenant au champ $effect$ de cet état.

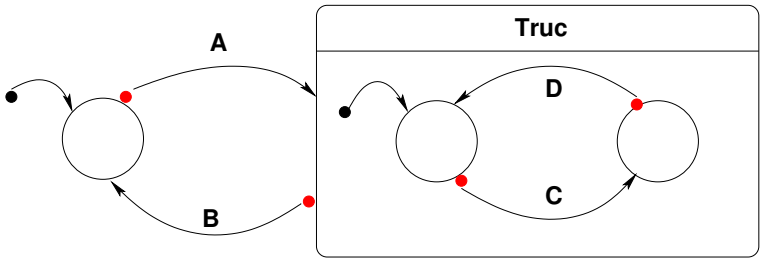

FIG. 3.9 – *Exemple de SyncChart contenant un macro-état*

La figure 3.9 page 64 donne un exemple de SyncChart contenant un macro-état. Considérons la constellation qui contient le macro-état *Truc* et sa traduction en graphe SynDEx. Le premier niveau de conditionnement correspondant à l'etat précédent (opération *CalculEtatCourant*) puis le deuxième correspondant au test du franchissement des arcs (opération(s) *FranchArcX*) définissent les 4 cas suivants :

- l'état de gauche est l'état précédent :
  - A est absent. L'état de gauche devient (reste) l'état courant (Cas 1);
  - A est présent. Le macro-état *Truc* devient l'état courant (Cas 2);
- Le macro-état *Truc* est l'état précédent :
  - B est absent. Le macro-état *Truc* devient (reste) l'état courant (Cas 3);
  - B est présent. L'état de gauche devient l'état courant (Cas 4).

Dans le cas 1, le macro-état *Truc* n'est ni l'état précédent, ni l'état courant, la constellation qu'il contient ne doit donc pas être évaluée et ne contient pas d'état courant. Dans le cas 4, puisque le macro-état *Truc* est préempté par un arc de préemption forte, la constellation qu'il contient n'est pas évaluée et si elle possédait un état précédent, il n'y a pas lieu de mémoriser celui-ci. Dans le cas 2, le macro-état *Truc* devient l'état courant, l'état initial de la constellation qu'il contient devient son état courant. Dans le cas 3 enfin, le macro-état *Truc* reste l'état courant et la constellation qu'il contient est à évaluer en fonction de son état précédent.

L'état courant de la constellation contenue par le macro-état *Truc* n'est donc indéterminé que dans le cas 3. Plus généralement, l'état courant d'une constellation contenue par un macro-état n'est à déterminer par une opération *CalculEtatCourant* que lorsque ce macro-état est l'état courant mais aussi l'état précédent. Ce n'est que dans ce cas que l'état courant de la constellation dépend de l'état précédent. Néanmoins, il est nécessaire de (ré-)initialiser l'état précédent lorsque le macro-état devient l'état courant alors qu'il n'était pas l'état précédent (Cas 2).

Il faut que l'état courant de la constellation contenue par le macro-état *Truc* calculé lorsque le cas 2 se produit soit disponible en tant qu'état précédent si le cas 3 se produit lors de l'instant logique suivant. L'opération retard $*Etat* de cette constellation devant être unique, il apparaît au

plus haut niveau de hiérarchie [3]. Puisqu'il consomme une donnée provenant d'un port de sortie d'une opération conditionnante, cela nécessite que pour tous les sous-graphes de cette opération (et des opérations conditionnantes qu'ils contiennent) une donnée soit produite pour ce port de sortie. Dans les sous-graphes correspondant à des cas où la constellation ne possède pas d'état courant (comme dans les Cas 1 et Cas 4 de notre exemple) on utilisera des constantes du type C_Etat_X pour retourner un entier qui puisse être consommé par l'opération retard. Dans les cas où le macro-état devient l'état courant alors qu'il n'était pas l'état précédent (Cas 2 de notre exemple), une constante dont la valeur correspond à l'état initial produit l'état courant qui deviendra, via l'opération retard, l'état précédent lors du prochain instant logique. Le port d'entrée correspondant à l'état précédent de cette constellation n'est utilisé que dans le sous-graphe correspondant au cas où, dans la constellation contenante, l'état précédent et l'état courant sont le macro-état (comme dans le Cas 3 de notre exemple).

L'arc initial d'une constellation et son état initial peuvent posséder un champ $effect$ non vide. Il convient donc, dans le cas où le macro-état contenant la constellation devient l'état courant, en plus d'initialiser l'état courant de produire les effets correspondants.

La figure 3.10 page 66 représente une opération $FranchArcX$ lorsqu'elle correspond à l'arc le moins prioritaire mais permet de passer d'un macro-état Tac à un macro-état Tic. Cette figure est à comparer à la figure 3.6 page 62 qui correspond au cas où l'arc le moins prioritaire relie deux états non hiérarchiques.

Le port d'entrée $a1$ de cette opération correspond à l'état précédent de la constellation contenue par le macro-état Tac, et le port de sortie $e1$ à son état courant. De même le port d'entrée $a2$ et le port de sortie $e2$ correspondent à l'état précédent et l'état courant de la constellation contenue par le macro-état Tic.

Dans le cas $c = 0$ où l'arc n'est pas franchi, le sous-graphe contient en plus l'opération conditionnante $CalculEtatCourantTac$ permettant de calculer l'état courant de la constellation contenue dans le macro-état source de l'arc. Cette opération est du type de celle décrite dans 3.2.3.1 page 58. Elle utilise la donnée du port $a1$ et produit l'état courant sur le port $e1$. La constellation contenue par le macro-état Tic n'étant pas évaluée, une opération constante produit une donnée pour le port $e2$. Afin de prendre en compte la possibilité que des mêmes signaux soit émis par les constellations contenues et la constellation contenante, une opération $ORforOUT$ réalise un *OR* booléen sur les sorties de même nom [4].

Dans le cas $c = 1$ où l'arc est franchi, on initialise l'état courant de la constellation contenue par le macro-état Tic grâce à l'opération C_Etat2_0 qui produit une donnée sur le port $e2$. La constellation contenue par le macro-état Tac n'étant pas évaluée, une opération constante produit une donnée pour le port $e1$. L'opération $EffectInitialTic$ produit les signaux correspondants aux champs $effects$ de l'arc initial et de l'état initial de la constellation contenue par le macro-état Tic. Comme dans le cas $c = 0$, une opération $ORforOUT$ réalise un *OR* booléen sur les sorties du même nom.

3. Il est en fait possible de trouver le niveau minimum auquel remonter l'opération retard, mais cela combiné à l'explication de ce calcul avec prise en compte de la préemption faible, suspension et arc instantané alourdirait considérablement le discours pour ce qui n'est en fait, qu'une optimisation d'implantation.

4. Cette opération est utilisée ici pour éviter de surcharger les graphes, mais on peut lui substituer autant d'opérations *OR* qu'il y a de signaux en sortie de la constellation contenue par le macro-état courant.

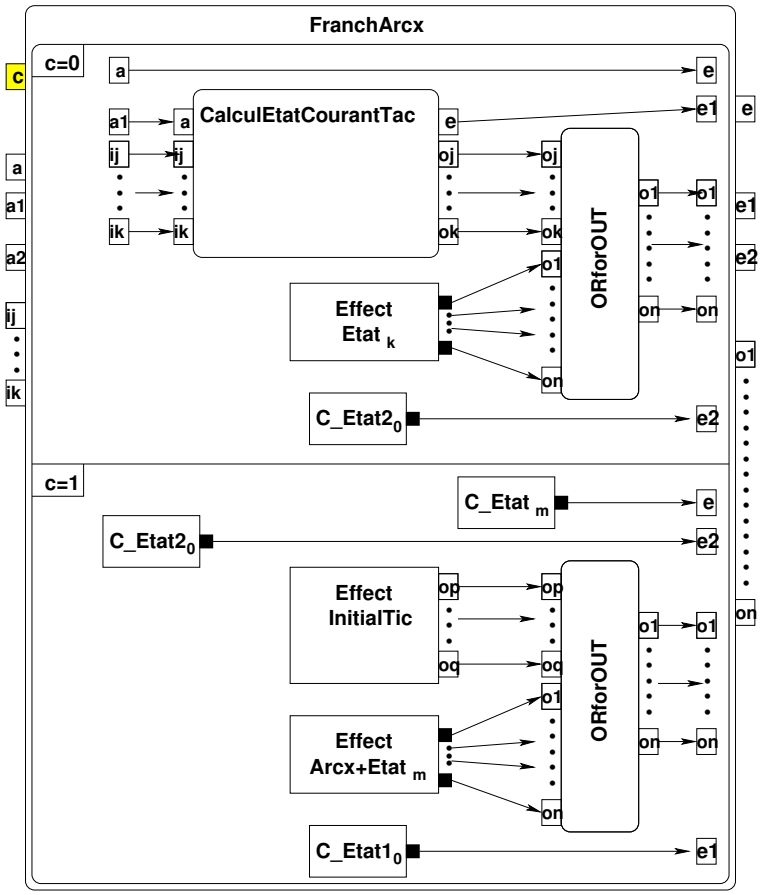

FIG. 3.10 – *Exemple d'opération FranchArcX traduisant le franchissement d'un arc de préemption forte entre deux macro-états*

Il se peut bien sûr que les macro-états utilisent la composition parallèle et contiennent plusieurs constellations. Le principe détaillé ici pour une est donc décliné plusieurs fois. Il peut ainsi y avoir plusieurs ports $a1$ et $e1$ pour les constellations préemptées et plusieurs ports $a2$ et $e2$ pour les constellations contenues par le macro-état destination. Les opérations retournant les constantes correspondant aux états et les opérations de type *CalculEtatCourant* sont alors aussi nombreuses que nécessaire.

3.2.3.4 Traduction des signaux locaux

Il est possible dans SyncCharts de spécifier des signaux locaux à un macro-état, c'est-à-dire pouvant être émis ou pouvant être utilisés dans les *trigger* des constellations contenues. La figure 3.11 page 67 donne deux exemples de programmes SyncCharts utilisant les signaux locaux. Il est possible, en utilisant les signaux locaux, de spécifier des programmes qui ne respectent pas la causalité : lors d'un instant logique, le statut de présence de tout signal doit être déterminé avant toute utilisation de ce signal. Cela nécessite l'existence d'une séquence d'évaluation du franchissement

des arcs qui permettent de statuer sur la présence d'un signal avant d'en avoir besoin. Cette règle est l'essence même de la sémantique constructive définie par G. Berry pour le langage Esterel [61].

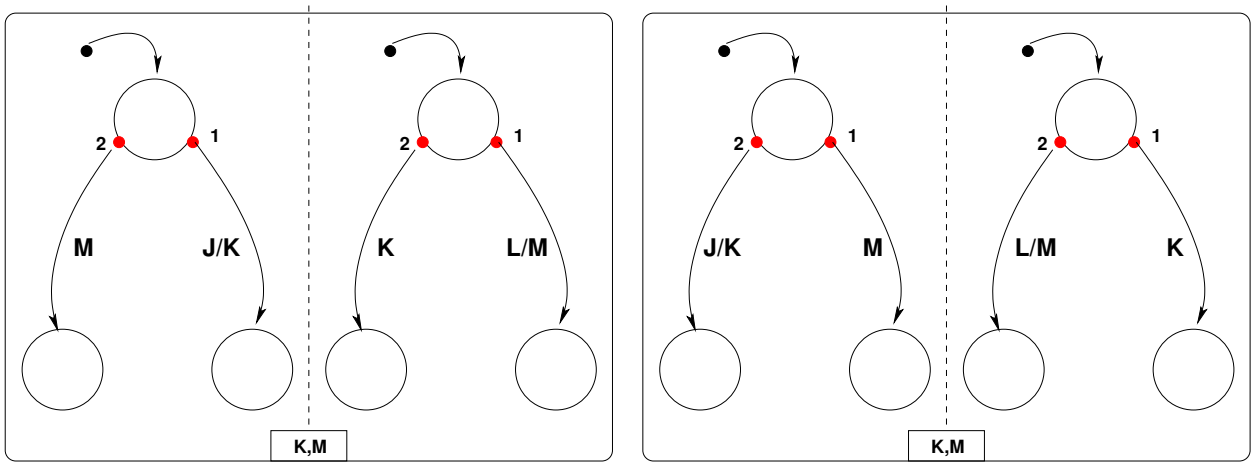

FIG. 3.11 – *Deux exemples de programme SyncCharts utilisant des signaux locaux*

Considérons l'exemple de gauche de la figure 3.11 page 67. Les signaux locaux K et M servent aux champs *trigger* des arcs les moins prioritaires des deux constellations. Après avoir évalué la transition la plus prioritaire de chaque constellation, on sait si K et M sont présents et donc il est possible d'évaluer, si besoin, les transitions moins prioritaires. Dans cet exemple, la causalité est respectée. Dans l'exemple de droite, par contre, il n'est pas possible de statuer sur la présence des signaux K et M et donc de décider du franchissement ou non des arcs les plus prioritaires. Considérer qu'ils ne sont pas franchissables peut mener, si les moins prioritaires le sont, à la présence de K et de M, ce qui serait en contradiction avec le non-franchissement des arcs les plus prioritaires. La présence de ce cycle montre que le programme ne respecte pas la causalité.

La traduction d'un programme SyncCharts en Esterel permet de vérifier la causalité du programme. Nous considérons donc que les programmes SyncCharts que nous traduisons respectent la causalité. Dans ce cas, la traduction est simple et consiste en la traduction d'une composition parallèle de constellations, mais où le port d'entrée d'une opération *CalculEtatCourant* peut être connecté à un port de sortie d'une opération *CalculEtatCourant* correspondant à une autre constellation. La figure 3.12 page 68 donne la traduction de l'exemple de gauche de la figure 3.11 page 67. Les dépendances de données semblent créer un cycle entre les deux opérations *CalculEtatCourant* mais la causalité du programme SyncCharts implique qu'une fois le graphe mis à plat, il n'en est rien.

3.2.3.5 Traduction de la préemption faible

Lorsqu'un arc de préemption faible est franchi :

- l'état courant de la constellation contenant cet arc devient l'état destination,
- si la source et/ou la destination de l'arc de préemption faible sont des macro-états, les constellations du macro-état source et du macro-état destination sont évaluées,

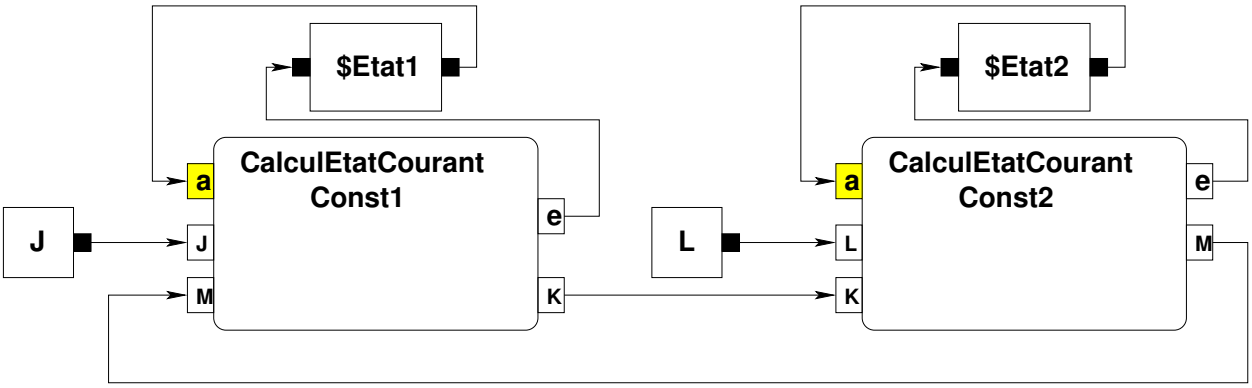

FIG. 3.12 – *Traduction du SyncChart de gauche de la figure 3.11 page 67*

– les signaux émis sont ceux des champs $effects$ de l'état source, de l'état destination, de l'arc franchi mais aussi des états courants et des arcs franchis dans les constellations contenues dans les macro-états source et destination.

Afin de traduire un arc de préemption faible on garde le principe des opérations $FranchArcX$ mais, lorsqu'elles correspondent à un arc de préemption faible, on modifie le sous-graphe correspondant au franchissement de l'arc ($c = 1$). Gardons l'exemple des deux macro-états Tac et Tic mais avec maintenant un arc de préemption faible allant de Tac à Tic. La figure 3.13 page 69 représente l'opération $FranchArcX$ correspondante où le sous-graphe correspondant au non-franchissement de l'arc n'est pas représenté car identique à celui de la figure 3.10 page 66. Nous rappelons toutefois que ce sous-graphe peut aussi prendre une autre forme si l'arc n'est pas l'arc le moins prioritaire permettant de quitter l'état source : il y alors une autre opération du type $FranchArcX$ dans ce sous-graphe (voir la figure 3.5 page 59).

Ces précisions faites, intéressons-nous à ce qui change et fait donc la particularité d'une opération $FranchArcX$ correspondant à l'évaluation du franchissement d'un arc de préemption faible (figure 3.13 page 69). En ce qui concerne la constellation qui contient Tac et Tic, l'opération C_Etat_m produit sur le port e l'entier correspondant à l'état courant, c'est-à-dire Tic. L'opération $EffectArcxEtat_m$ produit quant à elle les effets des champs $effect$ du macro-état Tic et de l'arc de préemption faible. En ce qui concerne la constellation contenue par le macro-état Tic qui devient l'état courant, l'opération C_Etat2_0 produit sur le port $e2$ l'entier correspondant à l'état initial de la constellation contenue dans Tic. L'opération $EffectInitialTic$ produit quant à elle les effets des champs $effect$ de l'état initial et de la transition initiale de cette constellation. Enfin, concernant la constellation contenue dans le macro-état préempté Tac, elle est évaluée une dernière fois grâce à l'opération $CalculEtatCourantTac$, qui utilise pour cela le port d'entrée $a1$ correspondant à l'état précédent de cette constellation. L'opération $ORforOUT$ permet de faire le consensus entre les signaux émis par $EffectArcxEtat_m$, $EffectInitialTic$ et $CalculEtatCourantTac$.

Le macro-état source d'un arc peut aussi être sa destination. Dans le cas où un tel arc est du type préemption faible, il y a réincarnation* de la ou des constellations contenues dans le macro-état. Le phénomène de réincarnation est similaire en programmation au phénomène de code ré-entrant. Une constellation est alors évaluée une dernière fois puis son état courant ré-initialisé, car dans

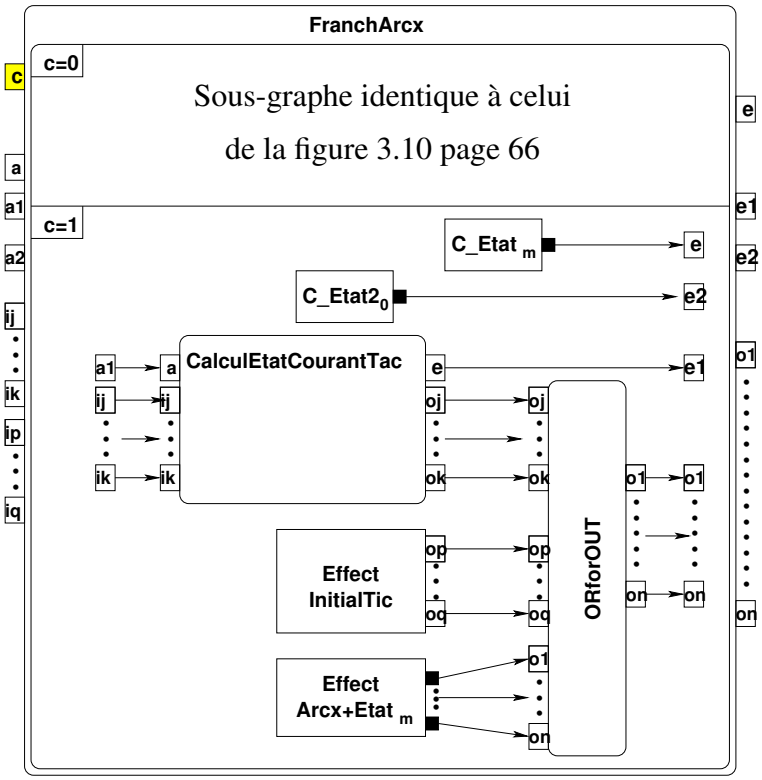

FIG. 3.13 – *Exemple d'opération FranchArcX traduisant le franchissement d'une transition de préemption faible entre deux macro-états*

la nouvelle incarnation c'est la transition initiale qui est franchie. Dans l'opération *FranchArcX* correspondant à l'arc de préemption faible, le graphe du cas où la transition est franchie diffère quelque peu de celui de la figure 3.13 page 69. En effet les ports d'entrée $a1$ et $a2$ sont confondus, ainsi que les ports de sortie $e1$ et $e2$. En conséquence, l'opération C_Etat1_k disparaît, et c'est l'opération *CalculEtatCourantTac* qui consomme l'état précédent alors que l'opération C_Etat2_0 produit l'état courant.

Ainsi, parce que les arcs de préemption forte sont plus prioritaires que les arcs de préemption faible, le sous-graphe d'une opération conditionnante *CalculEtatCourant* contient une imbrication d'opérations conditionnantes *FranchArcX*, d'abord de type préemption forte, puis de préemption faible.

3.2.3.6 Traduction de la terminaison normale

La figure 3.14 page 70 donne un exemple de SyncChart utilisant un arc de terminaison normale. Le macro-état *Truc* contient deux constellations et peut être quitté par un arc de préemption fort franchi si le signal E est présent ou par un arc de terminaison normale si les deux constellations contenues par *Truc* atteignent leur état final. Le signal F est alors émis.

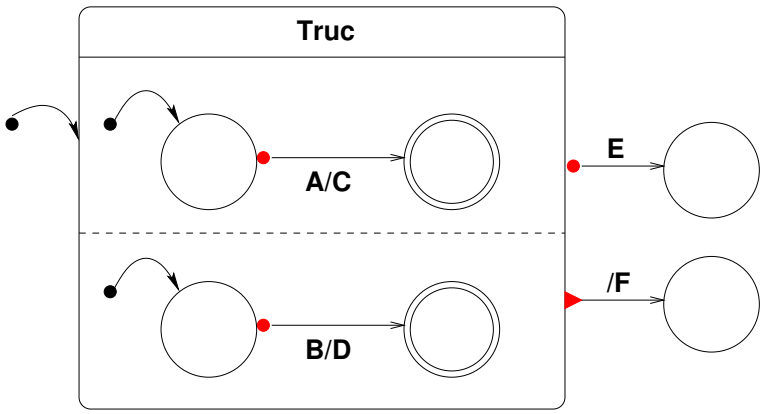

FIG. 3.14 – *Exemple de SyncChart contenant un macro-état pouvant être quitté par une terminaison normale*

Le principe de traduction de l'arc de terminaison normale repose sur plusieurs constatations :

- un macro-état ne peut être la source que d'un seul arc de terminaison normale. En effet chacune des constellations qu'il contient ne possède qu'un état final, et étant donné qu'un arc de terminaison normale ne possède ni de champ *trigger* ni de champ *condition*, il n'y a qu'un seul arc de terminaison normale dont le franchissement puisse être déclenché,
- l'arc de terminaison normale est le moins prioritaire des arcs ;
- avant de décider du franchissement de cet arc, il faut de toute façon évaluer les constellations contenues dans le macro-état source.

Jusqu'à présent dans notre traduction, le sous-graphe exécuté lorsque tous les arcs quittant un état ont été jugés non franchissables est celui correspondant au cas $c = 0$ du sommet $FranchArcX$ correspondant alors, à l'évaluation du franchissement de l'arc le moins prioritaire (voir figure 3.6 page 62 dans le cas où cet arc est un arc de préemption forte). S'il existe un arc de terminaison normale quittant l'état précédent, c'est ce sous-graphe qui est modifié comme le montre la figure 3.15 page 71. Celle-ci montre la structure de l'opération $Francharcx$ correspondant à l'arc de préemption forte préemptant $Truc$ dans le graphe de la figure 3.14 page 70, et dans le cas où il n'est pas franchi ($c = 0$), à la possibilité de franchir l'arc de terminaison normale.

La donnée de conditionnement de cette opération est le résultat du test sur la présence du signal E. Dans le cas où celui-ci est présent, c'est le sous-graphe $c = 1$ qui est exécuté. L'opération C_Etat_2 produit l'entier correspondant à l'état destination de l'arc de préemption forte. Cette donnée produite sur le port e correspond à l'état courant. Les constellations contenues dans le macro-état $Truc$ ne sont pas évaluées et l'opération C_Etat_2 produit un entier quelconque sur les ports $e1$ et $e2$ correspondant à leur état courant. L'opération $EffectArc1Etat_2$ produit les effets correspondant aux champs $effect$ de l'arc de préemption forte et de l'état destination.

Dans le cas où l'arc de préemption forte n'est pas franchissable, c'est le sous-graphe $c = 0$ qui est exécuté. Les opérations $CalculEtatCourantConst1$ et $CalculEtatCourantConst2$ correspondent à l'évaluation des constellations contenues par le macro-état $Truc$, et utilisent leur état précédent (ports d'entrée $a1$ et $a2$) et les signaux d'entrée A et B (ports d'entrée homo-

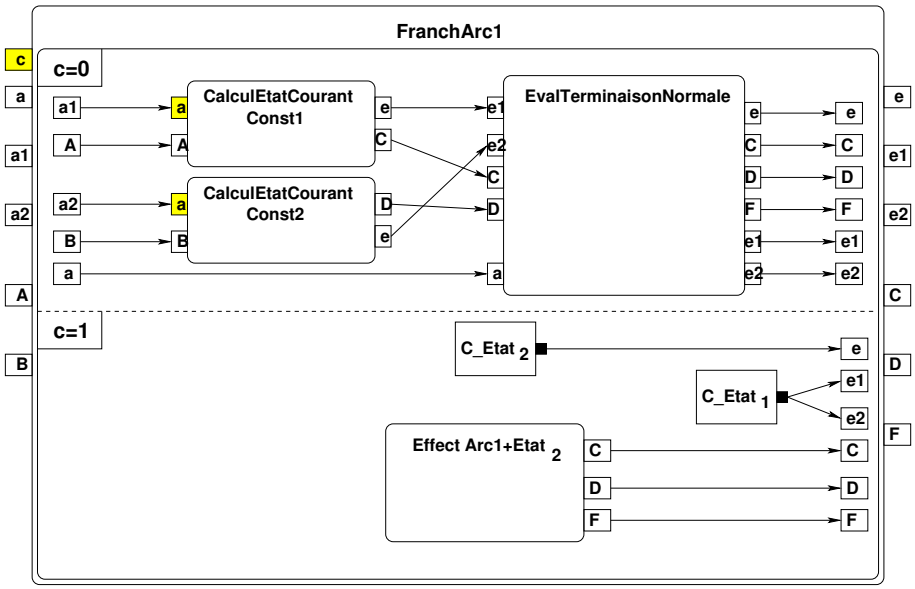

FIG. 3.15 – *Opération FranchArc1 appartenant à la traduction du SyncChart de la figure 3.14 page 70*

nymes). Elles produisent l'état courant de chacune de ces constellation (sur leur port *e*) et les données correspondant à la présence ou absence des signaux de sortie *C* et *D*. Mais au lieu de produire ces données directement sur les ports de sortie de l'opération *Franchar*1, c'est un opération *EvalTerminaisonNormale* qui les consomme.

Cette opération décide, en fonction des états courants calculés pour les constellations contenues dans le macro-état *Truc*, si l'arc de terminaison normale doit être franchi. Si c'est le cas, l'état courant de la constellation contenant le macro-état *Truc* devient l'état destination de l'arc de terminaison normale, ceux des constellations contenues dans le macro-état *Truc* deviennent quelconques, et aux effets de l'évaluation de ces constellations (émission possibles des signaux de sortie *C* et *D*) sont ajoutés les effets des champs $effect$ de l'arc de terminaison normale (ici émission du signal de sortie *F*) et de l'état destination de cet arc (ici ce champ est vide). Cet état destination peut en outre être un macro-état, et dans ce cas le sous-graphe doit aussi contenir l'initialisation des constellations contenues.

Dans le cas où l'arc de terminaison normale n'est pas franchi, les états courants des constellations contenues par le macro-état *Truc* sont inchangés, l'état courant de la constellation contenante aussi. Enfin un consensus entre les effets produits par les constellations contenues par le macro-état *Truc* et ceux du champ $effect$ de ce macro-état est effectué pour produire les signaux de sortie.

L'opération *EvalTerminaisonNormale* peut être décrite par un graphe. Dans celui-ci, une opération *TriggerTN* prend en entrée les états courant des constellations contenues par le macro-état source de l'arc de terminaison normale et produit une seule sortie valant 1 dans le cas où les états courants sont tous les états finaux, et 0 sinon. Cette sortie sert de donnée de conditionnement pour une opération conditionnante effectuant les actions décrites précédemment, suivant que l'arc de ter-

minaison normale est franchi ou non. Cette opération ressemble en fait beaucoup à une opération $FranchArcX$.

Nous avons ainsi traduit les 3 types d'arcs permettant de passer d'un état à un autre.

3.2.3.7 Traduction de la suspension

Un arc de suspension permet de geler les évolutions à l'intérieur d'un macro-état. Lorsque le macro-état est suspendu, cela signifie qu'aucun arc ne peut être franchi dans les constellations qu'il contient. Cela n'empêche pas le macro-état d'être préempté et n'annule pas les effets produits par les champs $effect$ du macro-état et des états courant des constellations qu'il contient.

Il s'agit donc dans le graphe SynDEx obtenu de n'exécuter l'opération *CalculEtatCourant* correspondant à une constellation que si celle-ci n'est pas suspendue. On définit alors une opération *Suspend* conditionnante dont la donnée de conditionnement est produite par une opération $TriggerArcX$ correspondant à la vérification de l'expression booléenne du champ $trigger$ de l'arc de suspension. Dans le cas où cette expression est fausse (macro-état non suspendu) le sous-graphe de l'opération *Suspend* ne contient que la ou les opérations *CalculEtatCourant* correspondant à la ou les constellations contenues par le macro-état.

Dans le cas où cette expression est vraie (le macro-état est suspendu) le sous-graphe consiste d'abord à laisser inchangés les états dans les constellations contenues dans le macro-état. Les ports d'entrée correspondant aux états précédents de ces constellations sont donc connectés aux ports de sortie correspondant aux états courants. D'autre part chacun des ports d'entrée correspondant à l'état précédent d'une constellation est connecté à une opération conditionnante $ConstEffect$ dont elle est la donnée conditionnante. Cette opération produit en sortie les signaux de sortie correspondant au champ $effect$ de l'état courant (ici égal à l'état précédent) de la constellation. Des opérations booléennes OR permettent de faire le consensus dans le cas d'émissions multiples d'un même signal.

3.2.3.8 Traduction de l'arc instantané

Les arcs instantanés permettent de traverser un état sans qu'il ne soit ni l'état précédent ni l'état courant. Le SyncChart de gauche de la figure 3.16 page 73 donne un exemple de constellation contenant un arc instantané. Si l'état précédent est l'état de gauche, et que seul le signal A est présent, l'état courant est celui du milieu. Mais si A et B sont présents l'état courant est l'état de droite. Le premier arc est d'abord franchi (présence de A), ce qui active l'état central. En absence d'arc instantané un état qui devient actif attend la présence strictement future des signaux permettant de franchir un arc dont il est la source. L'arc instantané annule la restriction "strictement future". Ici comme B est présent l'état central est quitté sans avoir été l'état courant. Ainsi les signaux C, D et E sont émis. Il est à noter que si l'arc instantané avait été un arc de préemption forte, les effets de l'état du milieu (le signal D) n'auraient pas été produit.

Considérons à présent le SyncChart de droite de cette même figure. Si l'état précédent est celui de gauche et que les signaux A et B sont présents, l'état courant est l'état du milieu. En effet, bien que l'arc franchi par la présence de A soit un arc instantané, cela n'empêche pas qu'une fois activé, l'état du milieu ne puisse être quitté que par la présence strictement future de B.

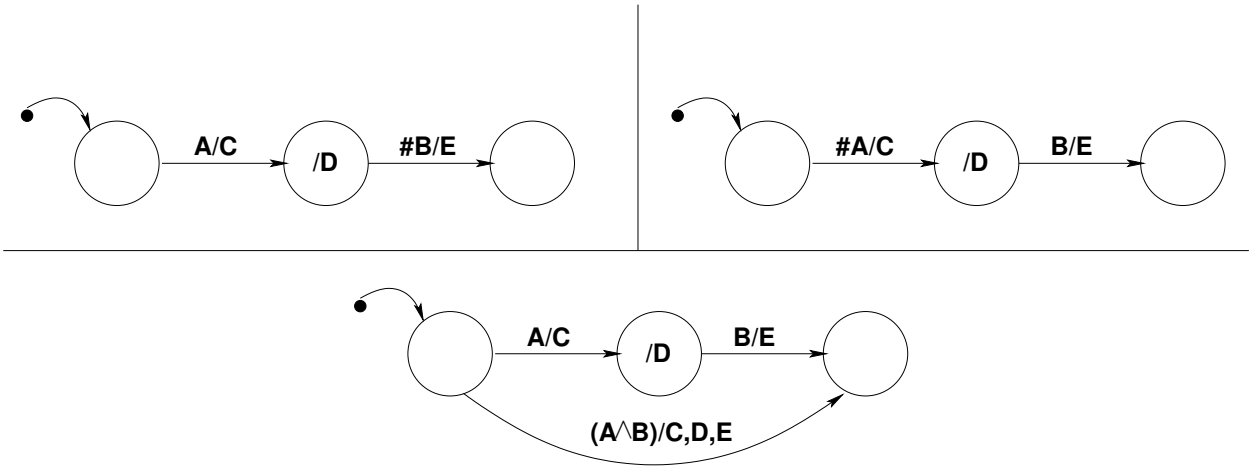

FIG. 3.16 – *SyncChart comportant des arcs instantanés et équivalent sans arc instantané*

Puisque notre traduction repose sur la définition de l'état courant en fonction de l'état précédent, on remarque que les arcs instantanés permettant de quitter l'état précédent n'ont aucune incidence. Ceux qui influent sur le comportement sont ceux qui permettent de quitter les états destinations des arcs ayant pour source l'état précédent. Afin de ne pas changer les principes de traduction précédemment énoncés, on transforme le SyncChart comportant des arcs instantanés en un graphe SyncCharts n'en comportant pas. Le SyncChart en bas de la figure 3.16 page 73 correspond au graphe obtenu par cette transformation à partir du SyncChart de gauche de la même figure.

La première étape de la transformation consiste à copier le SyncChart en remplaçant les arcs instantanés par des arcs normaux. Puis on parcourt le graphe d'origine, pour chaque état A :

- on considère chaque transition de type X (X pouvant être : préemption forte, préemption faible, terminaison normale) permettant de quitter cet état pour l'état B ;
- on cherche l'ensemble des états pouvant être atteints à partir de B en franchissant uniquement des arcs instantanés ;
- pour chaque état C de cet ensemble, on ajoute dans le graphe transformé un arc de type X ayant pour source A et pour destination source C. Son champ $Trigger$ est l'expression booléenne constituée d'un AND des champs $trigger$ de l'arc allant de A à B puis des arcs instantanés permettant d'aller de B à C [5] ;
- chaque arc créé possède également un champ $effect$ correspondant à une union des listes des signaux présents dans les champs $effect$ de l'arc allant de A à B, des arcs instantanés permettant d'aller de B à C, ainsi que de chaque état traversé si l'arc instantané qui le quitte n'est pas de préemption forte ;
- dans le cas où un état ainsi traversé est un macro-état et qu'il est quitté par un arc instantané

5. Il se peut qu'il existe plusieurs chemins d'arcs instantanés allant de B à C. Dans ce cas il y a autant d'arcs créés qu'il y a de chemins, chaque arc ayant alors un champ $trigger$ différent.

de préemption faible, il convient également d'ajouter les effets correspondant aux champs $effect$ de la transition initiale et de l'état initial de chaque constellation contenue par ce macro-état. Si ce dernier est lui aussi un macro-état, il faut renouveler l'opération avec les constellations qu'il contient, jusqu'à obtenir un état initial qui ne soit pas un macro-état. Si un de ces états initiaux est la source d'un arc instantané, il y a duplication de l'arc ajouté avec ajout sur la copie des effets de l'état destination et de cet arc instantané ;

– les arcs ajoutés sont, à type équivalent, plus prioritaires que les arcs d'origine (on ne peut refuser de franchir un arc instantané si son trigger est vrai). Les priorités entre deux arcs ajoutés de même type sont définis en comparant les priorités des arcs du graphe d'origine, auxquels les arcs ajoutés correspondent, en allant de A vers C.

Le graphe obtenu ne comportant plus d'arcs instantanés, il peut être ensuite traduit en utilisant les principes énoncés précédemment. Il existe malgré tout une exception : si l'état destination d'un arc de terminaison normale est la source d'un arc instantané, on obtient dans le graphe transformé un macro-état qui est la source de 2 arcs de terminaisons normales dont une possédant un champ *trigger*, ce qui d'après la sémantique SyncCharts n'est pas possible (voir 3.2.3.6 page 69). Notre traduction impliquant qu'il existe une seule opération $EvalTerminaisonNormale$ par macro-état, celle-ci doit évaluer quel arc de terminaison normale quittant ce macro-état va être franchi. Si comme nous l'avons proposé précédemment cette opération est décrite par un sous-graphe comportant une opération conditionnante, cette dernière devra inclure dans ses propres sous-graphes, une imbrication d'opérations $FranchArcX$ pour évaluer successivement le franchissement des différents arcs de terminaison normale, par ordre de priorité.

3.2.3.9 Signaux valués et opérations

Nous nous sommes jusqu'à présent restreints à des signaux non valués, simplement présents ou absents. Le franchissement d'un arc se faisait ainsi sur la vérification d'une expression booléenne portant sur la présence de signaux. De même les actions, déclenchées par un état ou le franchissement d'un arc, se limitaient à l'émission de signaux non valués.

La prise en compte des signaux valués se fait néanmoins naturellement. Pour la prise en compte du champ *condition* d'un arc, il suffit de modifier les opérations $TriggerArcX$ de la traduction pour qu'ils vérifient ces conditions. Pour ce qui est des effets (champs $effect$), les opérations $EffectArcXEtat_m$ et équivalentes[6] où sont modifiées pour produire non seulement la présence ou l'absence d'un signal, mais aussi sa valeur. Pour cela, une méthode communément employée par la communauté synchrone est de produire deux données par signal valué : un booléen pour coder l'absence ou la présence de ce signal, et un entier ou réel pour coder sa valeur. Cela modifiera la méthode présentée pour réaliser le consensus sur les signaux de sortie, qui pour l'instant utilisait des opérations booléennes OR. Celles-ci différeront selon la méthode spécifiée par l'utilisateur pour les émissions multiples. Cette méthode est appelée "fonction de combinaison" dans SyncCharts. Elle peut prendre par exemple la forme d'une somme ou d'un produit : un signal émis plusieurs fois au cours d'un instant logique prendra alors comme valeur la somme ou le produit

6. c'est-à-dire devant émettre les signaux correspondant au champ $effect$ d'un ou plusieurs (arcs initiaux) arcs et/ou d'un ou plusieurs (états initiaux) états.

des valeurs des différentes émissions.

Mais les effets peuvent ne pas se limiter à l'émission de signaux et prendre la forme d'appels à des fonction C par exemple. Là encore notre traduction s'adapte simplement, ces appels devenant des opérations contenues dans les opérations $EffectArcXEtat_m$ et équivalentes. Ces fonctions peuvent prendre en argument des données et les modifier. Ces données peuvent également être partagées par différentes fonctions présentes dans les champs $effect$ des différents états et transitions. L'implantation des graphes obtenus pouvant être distribuée, il n'est pas possible de garder le concept de diffusion instantané. Chaque entrée/sortie d'une fonction utilise alors une opération *retard* placée au plus haut niveau hiérarchique. Parce qu'il n'existe pas de règle dans SyncChart permettant de séquencer deux fonctions appartenant à des champs $effect$ différents, on fait l'hypothèse que le SyncChart est tel qu'il n'est pas possible qu'une donnée soit modifiée au cours d'un même instant logique par deux fonctions appartenant à des champs $effect$ différent. Afin d'appuyer cet hypothèse, reprenons l'exemple ABRO (voir figure 3.3 page 58). Si la transition dont le champ $trigger$ est A possède comme champ $effect$ l'instruction $x = x!$ et que la transition dont le champ $trigger$ est B possède comme champ $effect$ l'instruction $x = 2x$, le franchissement de ces deux arcs au cours du même instant logique mène à une valeur de x indéterminée [7] ce qui n'est pas acceptable.

3.2.3.10 Conclusion

L'approche proposée ici permet de traduire l'ensemble de la sémantique de SyncCharts. De nombreux langages proches de Statecharts ont une sémantique plus restreinte : moins de types d'arc, absence de suspension ou des arc instantanés, etc. Ces sémantiques, pour peu qu'elles soient assez strictes et déterministes, pourraient donc être traduites en appliquant les mêmes principes.

Ces principes sont différents de ceux que nous avions présentés dans l'article [62]. La principale différence est que la hiérarchie du SyncChart ne se traduisait pas par une hiérarchie de conditionnement dans le graphe SynDEx. Les opérations *CalculEtatCourant* des différentes constellations appartenaient toutes au niveau de hiérarchie le plus élevé. Mais chacune était incluse dans une opération conditionnante dont la donnée de conditionnement était l'état courant de la constellation contenante. Cette traduction générait des graphes SynDEx où de nombreux tests étaient inutiles : il fallait tester l'exécution de toutes les constellations, même si la constellation contenante n'était pas exécutée. Ce choix coûteux avait été fait parce que le modèle flot de données conditionné, et surtout sa version mise à plat, n'était pas complètement formalisé. Il ne permettait pas par exemple ce que nous appelons les faux-cycles, c'est-à-dire des dépendances de données formant un cycle à un certain niveau hiérarchique, comme c'est le cas sur la figure 3.12 page 68. La mise à plat, la substitution d'une opération par son sous-graphe, peut faire disparaître ce cycle si le sous-graphe n'est pas un graphe connexe utilisant toutes les entrées et toutes les sorties.

La disparition de cette contrainte lourde nous a permis de modifier notre traduction afin d'obtenir des graphes imbriquant les opérations conditionnantes plutôt que les séquençant, ce qui réduit les branchements conditionnels dans l'implantation finale. Cela nous a également permis de traduire les arcs instantanés et les arcs de préemption faible, qu'il était très difficile de traduire avec

7. $(2x)! \neq 2(x!)$

l'autre méthode [8].

3.3 Traduction Scicos/SynDEx

3.3.1 Le langage Scicos

Scicos* [9] [36] est une boîte à outils du logiciel libre de calcul scientifique Scilab [10], dédiée à la modélisation et la simulation de systèmes dynamiques hybrides.

Utilisant un modèle de schéma-bloc proche de celui de Simulink (dont il se veut une alternative libre), il permet de représenter des systèmes hybrides mêlant parties continues et discrètes. Pour les utilisateurs de Scicos, et les automaticiens en général, le terme "système" désigne le contrôleur (qui peut être un système au sens où nous l'entendons dans ce manuscrit, c'est-à-dire informatique) ainsi que l'environnement avec lequel il interagit. Par exemple, la description d'un système complet comprend souvent des blocs continus modélisant l'environnement physique, et des blocs discrets modélisant le contrôleur en interaction avec cet environnement. Une fois le système modélisé, son comportement est simulé par Scicos, permettant au développeur de visualiser des courbes correspondant aux différents signaux. Il est enfin possible de générer un code C monoprocesseur correspondant à la modélisation/simulation et pouvant s'exécuter indépendamment du simulateur.

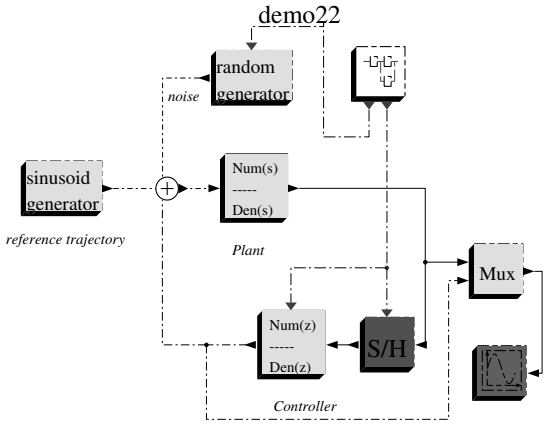

FIG. 3.17 – *Exemple de schéma-bloc Scicos*

Le modèle schéma-bloc utilisé communément pour décrire les systèmes hybrides dynamiques est proche du modèle flot de données, sauf que la règle d'activation n'est pas forcément respectée. Ainsi un arc représente un transfert de donnée entre deux blocs mais n'implique pas forcément

8. cela nécessitait en fait de modifier rétro-activement les états calculés précédemment, de dupliquer certaines opérations conditionnantes, menant ainsi à des implantations peu efficaces en nombre de tests et d'instructions

9. http://www.scicos.org

10. http://www.scilab.org

une relation d'ordre entre les exécution de ces deux blocs. Comme dans Simulink, les schémas-blocs Scicos permettent de décrire le conditionnement des blocs à l'aide de ports d'entrée spéciaux véhiculant une donnée booléenne conduisant à activer un bloc ou pas. Au terme opération utilisé précédemment pour le modèle flot de données, le modèle schéma-bloc substitue celui de bloc. Nous utiliserons donc celui-ci pour ne pas perturber les lecteurs plus familiers avec les schémas-blocs qu'avec le flot de données. La figure 3.17 page 76 montre un exemple de schéma-bloc Scicos.

3.3.1.1 Les signaux

Dans Scicos les systèmes sont modélisés sous la forme de schémas-blocs, les blocs sont une représentation graphique de fonctions à exécuter et sont reliés entre eux via leurs entrées et sorties par des arcs correspondant chacun à un signal. Il existe deux types différents de signaux :

- les signaux d'activation* (les entrées et les sorties sont situées au dessus et au dessous des blocs), correspondant à des arcs d'activation* servant au conditionnement,
- les signaux réguliers* (les entrées et les sorties sont situées à gauche et à droite des blocs), correspondant à des arcs réguliers* véhiculant les données consommées en entrée et produites en sortie par les fonctions que représentent les blocs.

Les signaux d'activation d'entrée (connectés aux ports situés sur le dessus du bloc) servent à conditionner l'exécution d'un bloc : si l'un de ces signaux vaut 1, la fonction associée au bloc est exécutée afin de produire, en fonction des signaux réguliers d'entrée (ports situés sur la gauche du bloc), les signaux réguliers de sortie (ports situés sur la droite du bloc) et les signaux d'activation de sortie (ports situés sur le dessous du bloc). On dit alors que le bloc est activé ou qu'il y a activation du bloc.

Tout signal peut être continu ou discret. Suivant que le signal d'activation activant un bloc est continu ou discret, on distingue deux cas :

- le signal d'activation est continu : il y a alors activation continue pendant un intervalle de temps (continu par morceaux). Par extension, on parlera de temps continu pour l'intervalle de temps considéré. Si un bloc est activé par des activations continues, son fonctionnement est continu ;
- le signal d'activation est discret : il y a donc activation discrète appelée événement, et on parlera alors de temps discret. Ces événements peuvent être générés à intervalles réguliers par une horloge pour obtenir une activation et donc une exécution périodique du bloc. Si un bloc continu est activé par des événements, alors son fonctionnement est discret.

Les signaux réguliers sont générés par des blocs activés via des signaux d'activation. A chaque signal régulier est donc associé un ensemble d'indices temporels appelés "périodes d'activation" pendant lesquelles le signal peut évoluer. En dehors de leurs périodes d'activation, les signaux restent constant comme le montre la figure 3.18 page 78. Un ensemble de périodes d'activation est une union d'intervalles temporels et de points isolés (événements). Un signal d'activation amène un bloc à évaluer chacune de ses sorties en fonction de ses entrées et de son (éventuel) état interne.

Un bloc n'ayant pas de port d'entrée d'activation hérite des périodes d'activation de l'union des périodes d'activation de ses signaux réguliers d'entrée. Lorsqu'un bloc peut être activé par

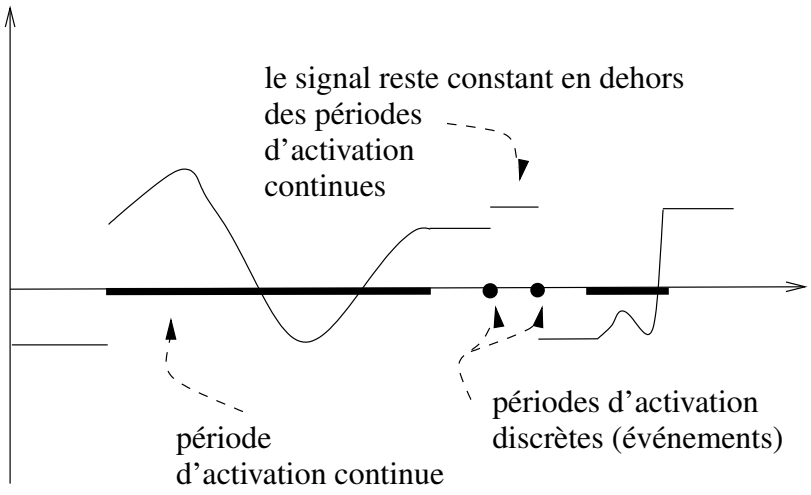

FIG. 3.18 – *Signal régulier dans Scicos et l'ensemble de périodes d'activation associées*

plusieurs signaux d'activation, on dira qu'il y a multi-activation de ce bloc. Deux représentations de ce bloc et de ces signaux d'activation sont alors permises :

- le bloc possède plusieurs entrées d'activation (sur le dessus) et au moins deux de ces entrées reçoivent un signal d'activation différent (exemple du bloc *A* de la figure 3.19 page 78). Le bloc est alors activé si l'un de ses signaux d'activation vaut 1, il est donc activé par l'union de ces signaux d'activation d'entrée ;
- l'union de plusieurs signaux d'activation est effectuée par un bloc "+" qui est ensuite connecté à l'une des entrées d'activation du bloc (exemple du bloc *B* de la figure 3.19 page 78).

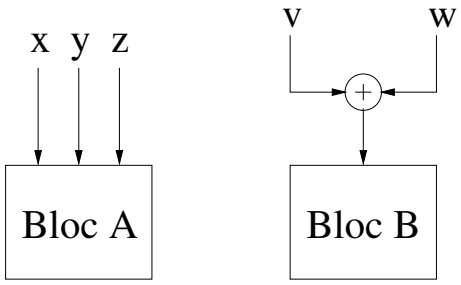

FIG. 3.19 – *Exemples de multi-activation*

3.3.1.2 Rémanence de donnée dans Scicos

Les arcs d'activation entrants indiquent pour un bloc donné, les blocs chargés de lui fournir le signal d'activation qui activera son exécution. A chaque activation correspond un instant logique

et une exécution du bloc. Cette exécution sera considérée comme synchrone [11] avec l'exécution des autres blocs activés par le même signal. Notons au passage que si certains blocs Scicos n'ont pas d'entrée d'activation, il en sera automatiquement créé au moins une lors de la compilation préalable à toute simulation [63].

Un arc régulier entre deux blocs n'assure en aucun cas que leurs exécutions soient synchrones. Il est donc possible qu'une opération puisse s'exécuter alors que ses entrées n'ont pas été produites lors de cette activation. Cela nécessite qu'il y ait rémanence des données sur les arcs réguliers : une donnée consommée reste disponible jusqu'à ce qu'elle soit remplacée par la production d'une nouvelle donnée par le producteur. Ceci diffère bien sûr du modèle flot de données où toute donnée produite ou consommée lors d'un instant logique n'est plus disponible lors du suivant (sauf utilisation explicite du sommet *retard*).

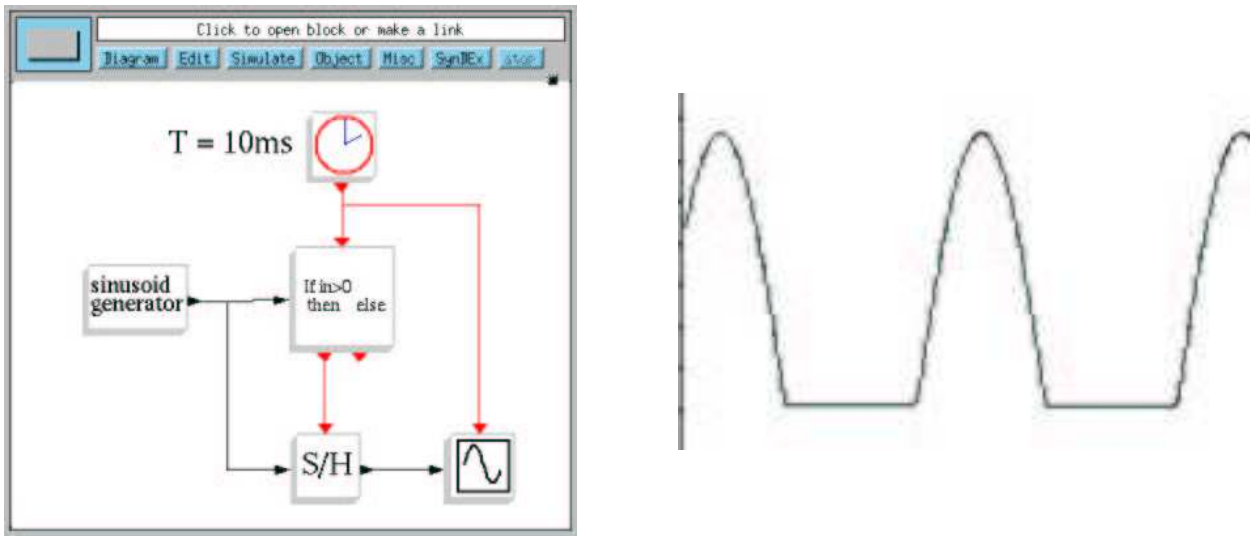

FIG. 3.20 – *Utilisation de la rémanence sous Scicos pour obtenir la partie positive d'une sinusoïde*

La partie gauche de la figure 3.20 page 79 présente la modélisation Scicos d'une application n'affichant que la partie positive d'une sinusoïde. Ceci est effectué via un bloc *sample and hold* (S/H) qui, lorsqu'il est activé, copie la valeur de son entrée sur sa sortie.

Le bloc *Clock* (représenté par un cadran d'horloge) génère un signal d'activation composé d'un ensemble d'évènements espacés de manière périodique, ici 10 ms. Le bloc *if then else* (*If in>0 then else*) produit des signaux d'activation sur ses sorties *then* (port inférieur gauche) et *else* (port inférieur droit). Lorsqu'il est activé, ici par un bloc *Clock* d'une période de 10 ms, le bloc *if then else* produit un signal d'activation :

– sur son port *then* si la valeur de son entrée régulière est positive. On dit alors que la branche *then* du bloc *if then else* est activée ;

11. au sens des langages synchrones où le temps d'exécution n'est pas considéré : la production des résultats d'un bloc est simultané avec la consommation des données. Des exécutions sont alors synchrones si elles ont lieu lors du même instant logique, correspondant ici à une activation discrète (événement).

– sur son port *else* sinon. On dit alors que la branche *else* du bloc *if then else* est activée.

L'entrée régulière du *if then else* étant fournie par la sinusoïde, le bloc *sample and hold* n'est activé que lorsque celle-ci est positive. Lorsqu'elle est négative et que le bloc d'affichage est activé (bloc en bas à droite), ce dernier prend la dernière valeur que le bloc *sample and hold* a fourni lors de sa dernière activation, disponible grâce à la rémanence.

Il est à noter que deux blocs appartenant à des branches différentes du bloc *if then else* ont des exécutions exclusives : ils ne peuvent être activés/exécutés en même temps.

3.3.1.3 Relations entre activations

Il convient ici de préciser quelques relations entre les activations.

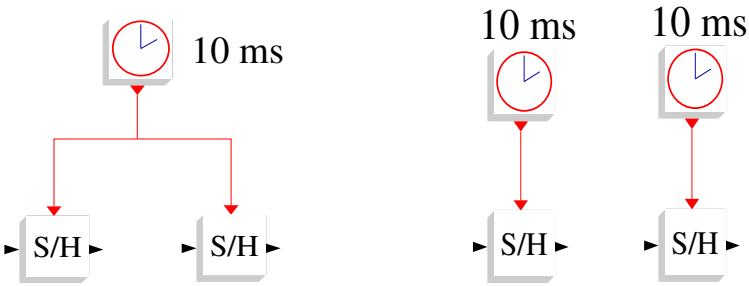

FIG. 3.21 – *Même période ne veut pas dire synchronisme*

Tout d'abord, Scicos fait une distinction entre les deux schémas-blocs de la figure 3.21 page 80 :

– dans le schéma-bloc de gauche deux blocs sont activés à l'aide de la même horloge (bloc *Clock*). Leurs exécutions sont donc synchrones ;
– dans le schéma-bloc de droite, deux blocs sont activés à l'aide de deux horloges distinctes générant des signaux à la même fréquence. Les activations de ces blocs ne sont pas synchrones car leurs signaux d'activation sont différentes.

Imaginons que dans les deux cas un arc régulier relie les deux *sample and hold*. S'ils sont synchrones, lors de la simulation, le producteur sera exécuté avant le consommateur. Dans le cas où ils ne le sont pas, l'ordre d'exécution est indéterminé.

Le bloc *if then else* permet de sous-échantillonner un signal d'activation en deux signaux dont l'intersection est vide et l'union égale au signal entrant (*i.e.*, exécution exclusive). Une généralisation du *if then else* existe via le bloc *event select* qui génère un signal d'activation sur une de ses sorties en fonction de la valeur d'entrée du signal régulier. Un comportement équivalent peut aussi être décrit avec plusieurs blocs *if then else*.

La figure 3.22 page 81 présente une méthode de sous-échantillonnage à l'aide d'un *if then else* dont le signal régulier est donné par un compteur modulo 2 activé par la même horloge que

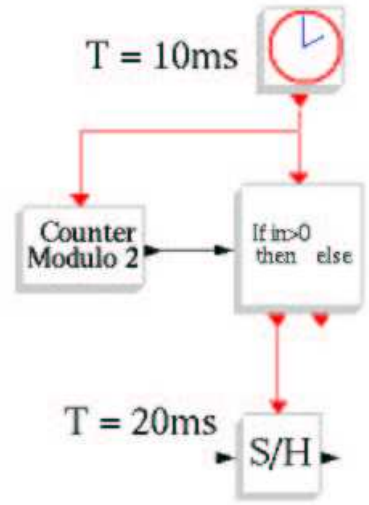

FIG. 3.22 – *Exemple de sous-échantillonnage*

le bloc *if then else*. Cela a pour conséquence de fournir au bloc *sample and hold* un signal dont la fréquence est divisée par 2 par rapport au signal original.

On étend la notion d'exécutions synchrones à la notion d'exécutions potentiellement synchrones*. On dira que deux blocs ont des exécutions potentiellement synchrones si leurs signaux d'activation sont des sous-échantillonnages d'un même signal. Sur l'exemple de la figure 3.22 page 81 les blocs *if then else* et *sample and hold* sont activés par des signaux provenant du même bloc *Clock* et leurs exécutions sont donc potentiellement synchrones.

Deux blocs aux exécutions exclusives peuvent avoir des exécutions potentiellement synchrones (schéma-bloc de gauche de la figure 3.23 page 81) ou pas (schéma-bloc de droite de la même figure).

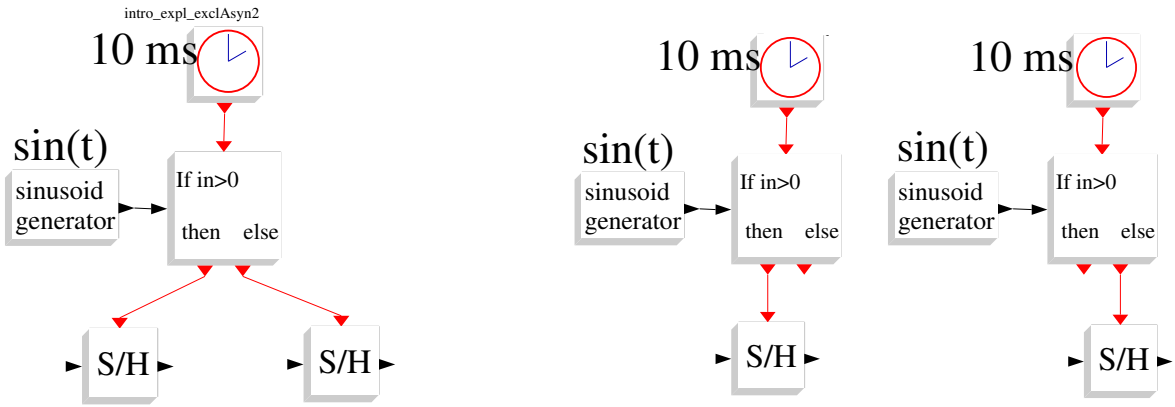

FIG. 3.23 – *Exclusivité et synchronisme potentiel*

3.3.2 Fonctionnalités additionnelles

3.3.2.1 Blocs sources et blocs puits dans une schéma-bloc Scicos

Lors de la description du système dans Scicos, l'utilisateur peut faire appel à des blocs "sources" afin de fournir des données en entrée de son programme (signal sinusoïdal, modulo N, lecture dans un fichier, etc.) ainsi qu'à des blocs "puits" afin de récupérer les données produites par le système (affichage écran, écriture dans un fichier, etc.). Lors d'une implantation temps réel, ces blocs correspondent respectivement à des capteurs et des actionneurs.

La figure 3.26 page 87 présente un système disposant de 3 capteurs et 1 actionneur.

3.3.2.2 "Region to superbloc"

Afin de faciliter la modélisation d'un sous-système, Scicos fournit une fonctionnalité visant, à partir d'une région du schéma-bloc sélectionnée par l'utilisateur, à créer un *superbloc* (bloc hiérarchique) contenant la partie du schéma-bloc sélectionnée, et créant les ports d'entrée et de sortie correspondants. Cela permet la description hiérarchique et la réutilisation d'une partie de la modélisation. La figure 3.24 page 82 représente l'utilisation de cette fonctionnalité afin de créer

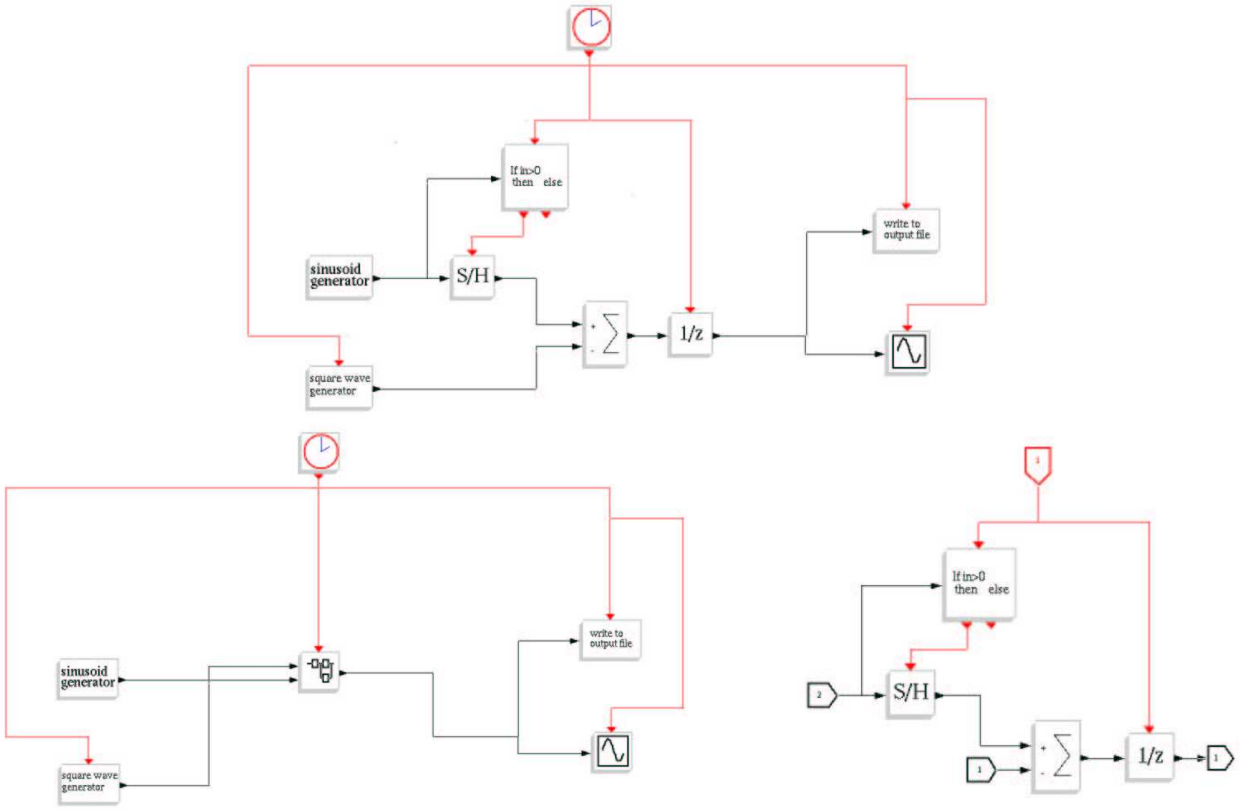

FIG. 3.24 – *Utilisation de "Region to superbloc" sous Scicos*

un *superbloc*. Le schéma-bloc Scicos en haut de la figure représente la modélisation sur un seul

niveau de hiérarchie. Le schéma-bloc en bas à gauche représente la même modélisation mais une partie du schéma-bloc a été remplacé par un *superbloc*. Le *superbloc* est celui situé en dessous du bloc *Clock*. La partie du schéma-bloc initial qu'il contient est donné par le schéma-bloc situé en bas à droite de la figure 3.24 page 82. Tout comme le *superbloc* ce schéma-bloc comporte une entrée d'activation, deux entrées régulières et une sortie régulière.

3.3.2.3 La pré-compilation des schémas-blocs Scicos

Avant la simulation d'un schéma-bloc Scicos, celui-ci subit une transformation appelée pré-compilation. Tout d'abord, il se peut que le développeur ait décrit un schéma-bloc non valide, par exemple comportant des cycles sans sommets *retard* (appelés aussi "boucles algébriques") ou bien dans lequel il manque des arcs entre les blocs (ports non connectés). Ensuite, afin de ne pas surcharger le schéma-bloc, certains blocs ne comportent pas d'entrée d'activation. Il faut pourtant savoir quand les exécuter. Enfin, de manière à gérer plus efficacement les signaux d'activation, il est préférable d'en constituer une hiérarchie afin de minimiser les tests.

Ces trois points sont gérés par la phase de pré-compilation des schémas-blocs Scicos. Les schémas-blocs non valides sont rejetés (boules algébriques, arcs manquants). Après pré-compilation, on obtient un schéma-bloc dont tous les blocs possèdent des entrées d'activation [12], déduites des activations de leurs prédécesseurs et/ou le cas échéant (cas des blocs sources) de leurs successeurs. Enfin, les blocs produisant des signaux d'activation sont parfois dupliqués afin d'établir une hiérarchie dans la production de ces signaux.

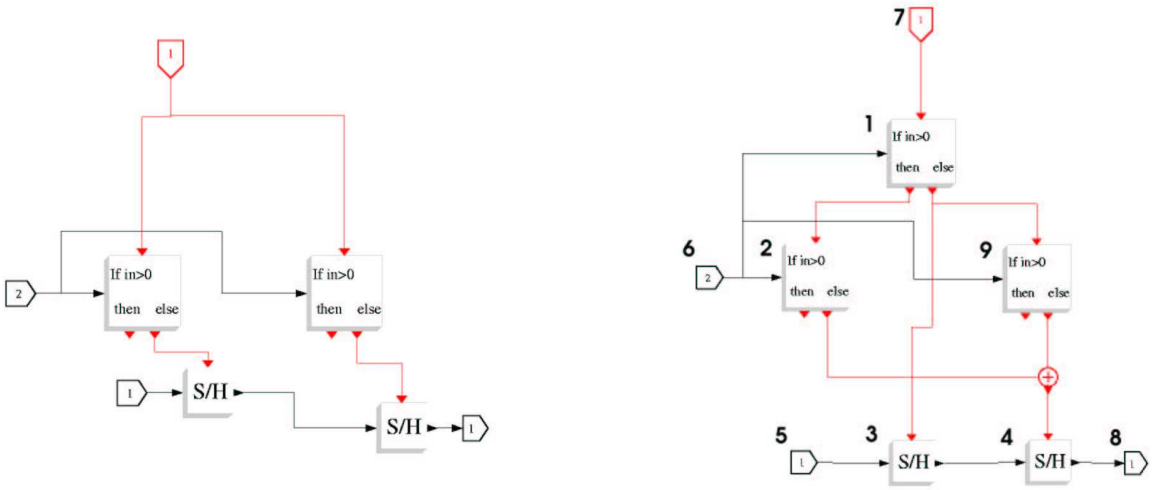

FIG. 3.25 – *La phase de pré-compilation Scicos*

La figure 3.25 illustre, avec un schéma-bloc simple, cette construction d'une hiérarchie entre les blocs produisant les signaux d'activation. Le schéma-bloc de gauche est celui tel qu'il a été décrit

12. souvent en ajoutant des blocs "+" dans les cas de multi-activation, comme expliqué page 78.

par le développeur, celui de droite est celui obtenu après pré-compilation. Si l'on compare les deux schémas-blocs on a tout d'abord l'impression que la pré-compilation "complique" le schéma-bloc initial : apparition d'un bloc *if then else* supplémentaire et apparition d'un bloc "+" synonyme de multi-activation. Afin de définir les différents cas d'activation possibles entre les blocs connectés par des arcs réguliers, la pré-compilation crée des relations de sous-échantillonnage entre des blocs producteurs de signaux d'activation n'en ayant pas au départ.

Ici, les deux blocs *if then else* n'ont pas au départ de relations entre eux (schéma-bloc de gauche de la figure 3.25). La pré-compilation remplace celui de droite par deux blocs *if then else* (duplication) qui sont placés dans chacune des branches *then* et *else* de l'autre bloc *if then else*. Le comportement est conservé car le deuxième bloc *if then else*, par la duplication, est tout de même toujours activé quand le premier l'est, que ce soit par sa branche *then* ou sa branche *else*. La différence est qu'il est dorénavant possible de définir une relation de sous-échantillonnage entre les blocs *if then else*, ce qui facilitera la traduction des arcs réguliers.

La duplication est visible sur la figure 3.25 page 83 :

- dans le schéma-bloc de droite, le bloc *if then else* numéro 1 correspond au bloc *if then else* de gauche du schéma-bloc initial,
- le deuxième bloc *if then else* du schéma-bloc de gauche est quant à lui dupliqué de manière à ce qu'une de ses instances (bloc numéro 2) soit dans la branche *then* du bloc *if then else* numéro 1 et que l'autre instance (bloc numéro 9) soit dans sa branche *else*.

Un bloc "+" apparaît pour que le bloc *sample and hold* de droite soit activé par les deux instances du bloc *if then else* initial. Ce bloc "+" réalise une union des signaux d'activation qu'il reçoit (multi-activation). Dans le schéma-bloc obtenu après pré-compilation, on peut ainsi en comparant le signal d'activation du premier *sample and hold* avec les deux signaux d'activation du deuxième *sample and hold*, considérer les différents cas possibles d'activation des ces deux blocs.

La pré-compilation peut ainsi générer des cas de multi-activation alors que le schéma-bloc original n'en comporte pas. Néanmoins la phase de pré-compilation garantit d'une part que les signaux d'activation unis par un bloc "+" sont exclusifs deux à deux, et d'autre part qu'aucun bloc producteur de signaux d'activation n'est multi-activé.

Adoptant la même démarche qu'avec le langage SyncCharts, notre traduction n'est destinée qu'à des schémas-blocs valides. De plus, celle-ci nécessite de connaître les différents cas possibles d'activation des blocs connectés par des arcs réguliers, c'est pourquoi nous utiliserons le schéma-bloc obtenu après la pré-compilation comme entrée de notre traduction.

3.3.3 Principes de la traduction

3.3.3.1 Divergences principales entre les langages

Il s'agit ici d'exprimer les principales divergences entre le langage Scicos et le langage SynDEx.

Si tous les blocs composant un schéma-bloc Scicos sont activés par le même signal d'activation, il est possible de traduire le schéma-bloc Scicos en programme flot de données homogène (non conditionné) comme l'ont montré les auteurs de [64]. La difficulté apparaît lorsque le schéma-

bloc Scicos standard comporte des blocs reliés par des arcs réguliers et consommant des signaux d'activation différents.

Tout d'abord cela introduit de la rémanence qui, implicite dans le schéma-bloc Scicos, doit être explicitée dans le programme flot de données. Chaque cas d'arc régulier doit être étudié précisément car il ne correspond pas forcément à une dépendance de données dans un programme flot de données. En effet une dépendance de données implique la production d'une donnée avant sa consommation. Si production et consommation n'ont pas lieu lors du même instant logique, un sommet *retard* doit être employé mais cela implique alors qu'il y aura toujours un décalage d'un instant logique entre production et consommation.

Dans un graphe SynDEx, l'ordre (séquence) est induit par la règle d'activation (ordre entre les opérations) et le conditionnement par l'utilisation des opérations conditionnantes. Dans un schéma-bloc Scicos, le contrôle est moins explicite : les arcs réguliers n'impliquent pas toujours un ordre (cas de rémanence) et le conditionnement apparaît après étude des arcs d'activation et surtout des relations entre eux (sous-échantillonnage, exclusion). C'est justement le but du simulateur d'en extraire le contrôle afin de déduire une séquence d'exécution des différents blocs et des signaux réguliers correspondants. Les arcs réguliers et les arcs d'activation ne sont donc pas traités de la même manière. On remarque également que le conditionnement dans Scicos utilise des arcs pour définir les blocs étant activés ensemble, alors que le langage SynDEx utilise pour cela la hiérarchie, l'appartenance à un même sous-graphe. Si la sémantique d'une opération conditionnante est fixe, les blocs qui produisent des signaux d'activation dans Scicos peuvent utiliser des fonctions différentes, ce qui modifie la façon dont ces signaux d'activation sont générés : le bloc *Clock* possède par exemple une fréquence définie par le développeur. L'expression du conditionnement et du contrôle en général est donc moins déductible du graphe (c'est-à-dire sans descendre au niveau des fonctions attachées à chaque sommet) que dans le cas du langage SynDEx.

Ces remarques impliquent quelques restrictions sur les schémas-blocs Scicos qui pourront être traduits en flot de données conditionné.

3.3.3.2 Restrictions sur les schémas-blocs pouvant être traduits

Scicos permet de modéliser un contrôleur et son environnement afin de vérifier grâce à la simulation le comportement de ce contrôleur. Un fois la modélisation satisfaisante, seul le contrôleur est implanté sur une architecture matérielle, constituant alors un système au sens où nous l'entendons.

Notre traduction ayant pour but final d'obtenir cette implantation, nous ne traduirons que la partie contrôleur des schémas-blocs Scicos. Celle-ci pourra par exemple être encapsulée dans un *superbloc*.

Ce *superbloc* peut contenir des bloc standards, c'est-à-dire ayant une ou plusieurs entrées et sorties régulières, aucune ou une seule entrée d'activation et aucune sortie d'activation.

Notre implantation distribuée devant être déterministe afin de pouvoir satisfaire des contraintes temps réel strict, il convient de rejeter tout schéma-bloc non déterministe. Nous avons vu à ce titre qu'un arc régulier entre deux blocs aux exécutions non potentiellement synchrones donnait lieu à un ordre non déterminé d'exécution : un schéma-bloc comportant un tel arc ne pourra donc pas être traduit.

Plus encore, le cas de blocs aux exécutions non potentiellement synchrones part du principe

qu'il existe plusieurs signaux d'activation sans relation entre eux. C'est la simulation qui fixe arbitrairement une relation entre ces signaux d'activation. Si on reprend l'exemple de la figure 3.21 page 80, la simulation peut indifféremment considérer qu'il existe une phase de 2, 4 ou même 8ms entre le déclenchement des signaux d'activation des deux blocs *Clock*. L'absence de déterminisme implique que la traduction ne peut s'appliquer que si tous les blocs du schéma-bloc ont une exécution potentiellement synchrone [13].

Enfin dans un graphe SynDEx, il n'existe qu'une base de temps, c'est la répétition infinie du graphe. Toutes les exécutions des opérations se font à ce même "rythme" ou, grâce aux opérations conditionnées, à un sous-échantillonnage de ce "rythme".

Il faut donc au final que le *superbloc* ne comporte qu'un seule entrée d'activation et aucun bloc *Clock* (pour ne pas entraîner d'exécutions non potentiellement synchrones). Il peut par contre contenir des blocs producteurs de signaux d'activation à la condition que ceux-ci possèdent une et une seule entrée d'activation, une ou plusieurs entrées régulières, aucune sortie régulière et une ou plusieurs sorties d'activation. Il est dans ce cas possible d'exprimer un tel bloc producteur de signaux par un schéma-bloc Scicos n'utilisant que des blocs standards et des blocs *if then else*.

La figure 3.26 page 87 montre un schéma-bloc contenu dans une *superbloc* et qui satisfait les restrictions précédentes.

3.3.3.3 Principes de base

Nous décrivons ici la transformation du schéma-bloc en prenant en compte uniquement les fonctions attachées aux blocs producteurs de signaux car elles sont génératrices de conditionnement. Les fonctions attachées aux blocs standards ne sont pas prises en compte car ne posent pas de problème théorique : une fonction attachée à un bloc consommant 4 entrées et réalisant un traitement pour produire 3 sorties devient une fonction attachée à une opération consommant 4 entrées et réalisant un traitement pour produire 3 sorties. Nous détaillerons néanmoins plus loin dans le manuscrit un traducteur complet de schéma-bloc Scicos basé sur les principes présentés ici.

L'idée principale de la traduction est que les blocs standards sont traduits par des opérations, les arcs réguliers sont traduits en dépendances de données avec, parfois, ajout d'une opération *retard*, mais que les arcs d'activation et les blocs producteurs de signaux d'activation disparaissent, remplacés par des opérations conditionnantes.

Par exemple, un bloc Scicos *if then else* est traduit par une opération conditionnante possédant deux sous-graphes : un pour représenter la branche *then* et un pour la branche *else*. C'est ce qu'illustre la figure 3.27 page 88, avec à gauche une vue schématisée d'un *if then else* dans Scicos et à droite les entités flot de données conditionné correspondantes. A noter que l'opération > 0 est ajoutée afin d'expliciter le test que fait le bloc *if then else* sur la donnée reçue sur son entrée régulière : cette opération produit la valeur 1 si l'entrée est supérieure à 0 et 2 sinon. Les blocs se trouvant dans les branches A et B du schéma-bloc Scicos trouvent leur traduction dans les sommets constituant les sous-graphes A et B (représentés par des "nuages" sur le graphe de droite de la figure 3.27 page 88) du graphe SynDEx.

L'ensemble des blocs activés par un signal d'activation est donc traduit par un sous-graphe

13. ce qui inclut bien sûr les exécutions synchrones.

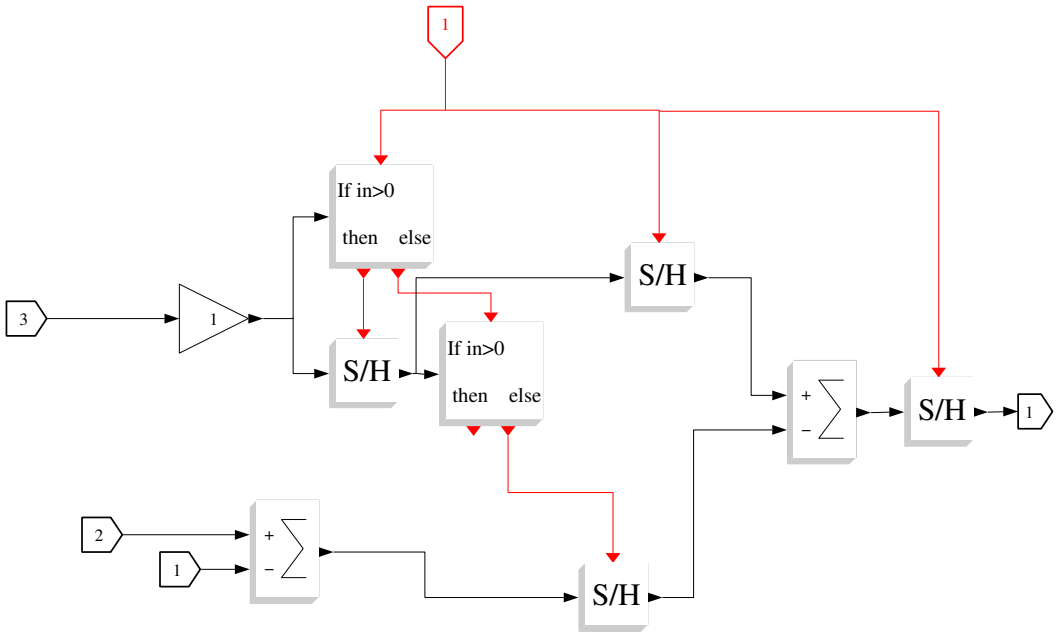

FIG. 3.26 – *Exemple de schéma-bloc dont tous les blocs ont des exécutions potentiellement synchrones et donc traduisible en graphe SynDEx*

d'une opération conditionnante correspondant à la source de ce signal d'activation. A un bloc standard correspond donc une opération comportant un nombre de ports d'entrée égal au nombre d'entrées régulières du bloc et un nombre de ports de sortie égal au nombre de sorties régulières. Les entrées d'activation, quant à elles disparaissent, remplacées par l'appartenance à un sous-graphe d'une opération conditionnante.

La figure 3.28 page 89 donne un exemple de schéma-bloc Scicos comportant un bloc "Counter Modulo 3" activé par la branche *then* d'un bloc *if then else*, ainsi que le graphe SynDEx correspondant. A noter qu'il manque évidemment des arcs réguliers sur le schéma-bloc Scicos (au niveau du bloc *sample and hold*) pour en faire un schéma-bloc correct : c'est voulu pour cet exemple, la traduction des arcs réguliers étant détaillée plus loin.

Dans Scicos, un bloc peut être activé par un signal d'activation qui correspond à l'union de plusieurs signaux d'activation (via le bloc "+"). Nous rappelons qu'après la pré-compilation, les signaux d'activation unis par un tel bloc sont forcément exclusifs. Lorsqu'un bloc est multi-activé, le graphe SynDEx correspondant comporte autant d'instances de l'opération correspondante qu'il existe de signaux d'activation exclusifs connectés au bloc, que ce soit directement ou par l'usage du bloc "+". Ces instances sont placées dans les sous-graphes correspondant aux différents signaux d'activation.

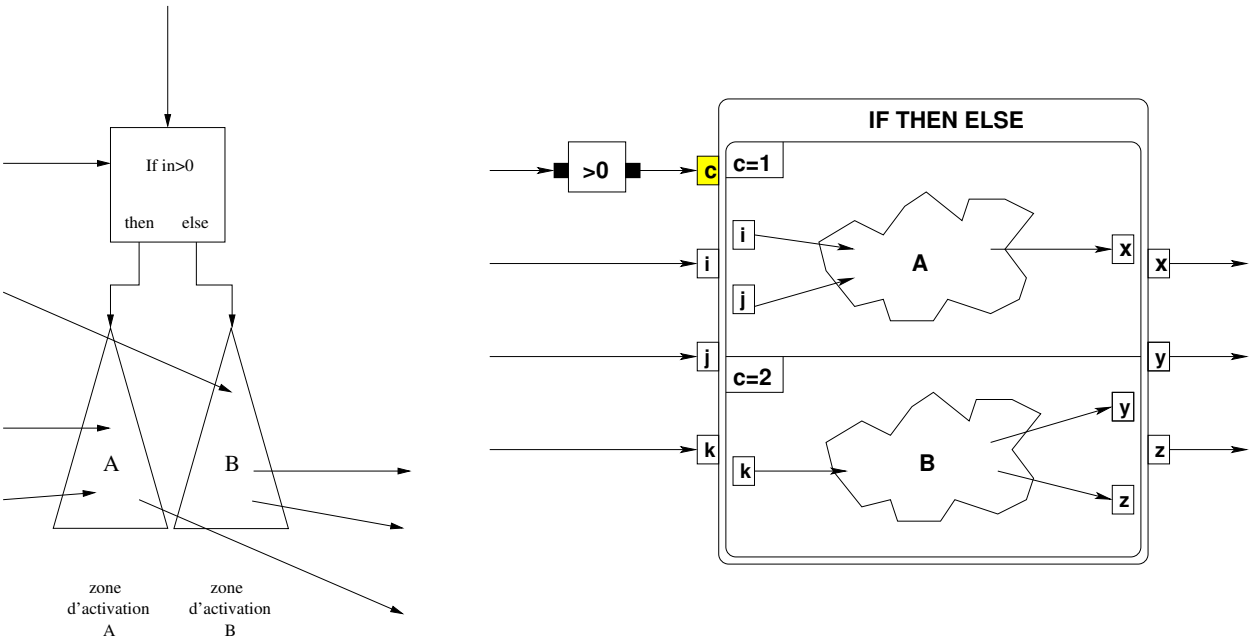

FIG. 3.27 – *Traduction d'un if then else en une opération conditionnante*

A gauche de le figure 3.29 page 89 se trouve un exemple de schéma-bloc Scicos où un bloc "Counter Modulo 3" est multi-activé par l'union des deux branches d'un *if then else*. A droite de la même figure, le graphe SynDEx équivalent comporte deux opérations "Counter Modulo 3", une dans chaque sous-graphe de l'opération conditionnante correspondant au bloc *if then else*.

Un bloc *if then else* pouvant activer un autre bloc *if then else* par une de ses branches, l'opération conditionnante correspondant à un bloc *if then else* peut contenir dans un de ses sous-graphes une opération conditionnante correspondant à un autre bloc *if then else*. La base de la traduction est donc la traduction des activations du schéma-bloc Scicos pré-compilé par une hiérarchie d'opérations conditionnantes contenant les instances, multiples en cas de multi-activation, de chaque opération correspondant à un bloc standard. Il reste ensuite à traduire les arcs réguliers qui nécessitent des règles plus détaillées, notamment à cause de la rémanence.

La figure 3.30 page 90 montre à droite la traduction des activations du schéma-bloc [14] Scicos pré-compilé visible à gauche de la même figure.

3.3.3.4 Cas simple

Lorsque tous les bloc contenus par un *superbloc* sont activés par un seul et unique signal d'activation, chaque arc régulier est traduit par une dépendance de données. La figure 3.31 page 90 montre la traduction d'un tel *superbloc* en graphe SynDEx. A noter que c'est dans ce cas seulement que la forme du schéma-bloc initial est conservée.

14. déjà utilisé dans la figure 3.25 page 83 pour illustrer la pré-compilation

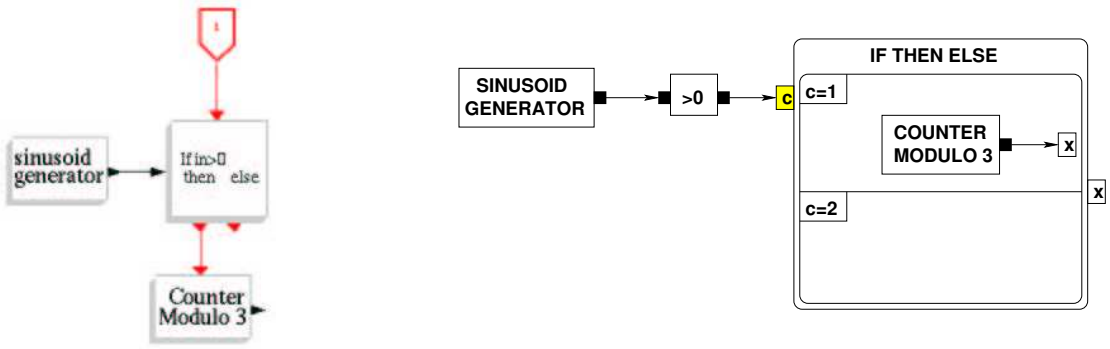

FIG. 3.28 – *Traduction d'un bloc if then else et d'un bloc sample and hold en flot de données conditionné*

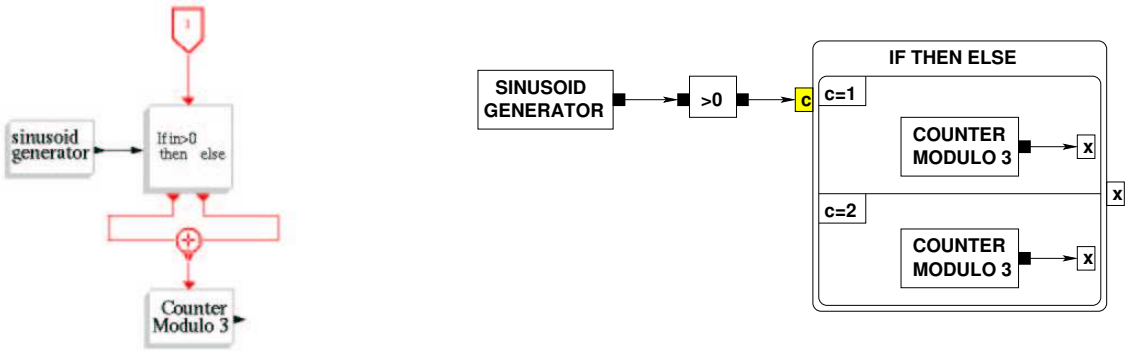

FIG. 3.29 – *Traduction d'un bloc if then else et d'un bloc sample and hold multi-activé en flot de données conditionné*

3.3.3.5 Caractérisation des relations entre signaux d'activation

La traduction d'un arc régulier reliant un bloc producteur à un bloc consommateur dépend des relations existant entre leurs signaux d'activation. Soit deux signaux d'activation s_i et s_j, on définit les notations suivantes :

- $s_i = s_j$ signifie que s_i et s_j sont un seul et même signal d'activation ;
- $s_i \, S_{ech} \, s_j$ signifie que s_i est un sous-échantillonnage de s_j. De même, $s_i \, S_{echeq} \, s_j$ signifie que soit s_i est un sous-échantillonnage de s_j, soit $s_i = s_j$;
- $s_i \, X_{cl} \, s_j$ signifie que s_i et s_j sont exclusifs.

On nomme A_P l'ensemble des signaux d'activation activant le producteur et A_C l'ensemble des signaux d'activation activant le consommateur. Ainsi, si on généralise aux signaux composant les ensembles A_P et A_C, on distingue cinq cas :

- $A_P = A_C$: les ensembles sont identiques, producteur et consommateur sont toujours activés ensemble,

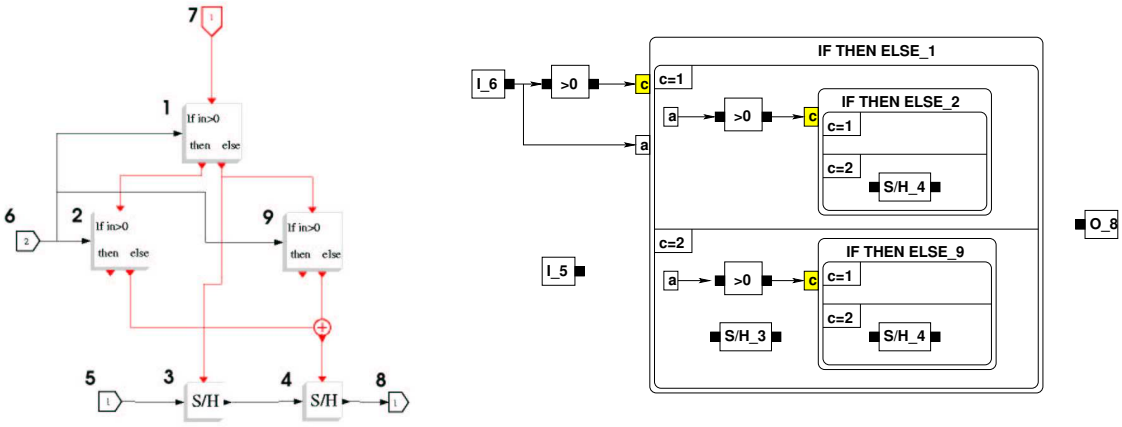

FIG. 3.30 – *Traduction des activations d'un schéma-bloc Scicos en hiérarchie d'opérations conditionnantes*

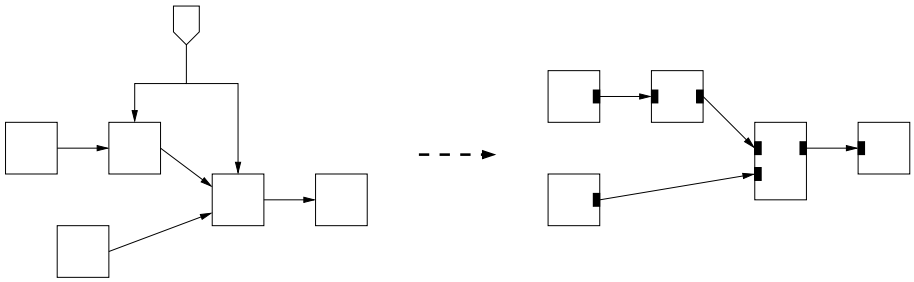

FIG. 3.31 – *Traduction d'un schéma-bloc Scicos ne possédant qu'un signal d'activation*

- $A_P \neq A_C$ et $\forall s_j \in A_C, \exists s_i \in A_P \mid s_j \; S_{echeq} \; s_i$: le producteur est toujours activé quand le consommateur l'est, mais peut aussi être activé seul. On notera ce cas $A_C \subset A_P$;
- $A_P \neq A_C$ et $\forall s_i \in A_P, \exists s_j \in A_C \mid s_i \; S_{echeq} \; s_j$: le consommateur est toujours activé quand le producteur l'est, mais peut aussi être activé seul. On notera ce cas $A_P \subset A_C$;
- $\forall s_i \in A_P, \forall s_j \in A_C, s_i \; X_{cl} \; s_j$; si le consommateur est activé le producteur ne l'est pas et réciproquement. On notera ce cas $A_C \cap A_P = \emptyset$;
- aucun des cas précédent : le producteur peut être activé seul, le consommateur aussi, ou ils peuvent être activés ensemble. On appellera ce cas le *cas mixte* car les ensembles sont alors composés de signaux combinant inclusion dans un sens et dans l'autre et/ou l'exclusion.

A chaque cas correspond une règle permettant de traduire l'arc régulier correspondant. Le cas simple présenté précédemment correspond au cas $A_P = A_C$ sans multi-activation.

3.3.3.6 Hypergraphe acyclique orienté d'activation

Afin de déterminer pour deux blocs la relation existant entre leurs signaux d'activation, on déduit du schéma-bloc pré-compilé Scicos un Hypergraphe Orienté Acyclique d'Activation (HOAA)* qui exprime la hiérarchie des signaux d'activation (relations de sous-échantillonnage). Ce HOAA ne possède qu'un sommet source (sans prédécesseur) correspondant à l'entrée d'activation du *superbloc* traduit. Chaque hyperarc représente la relation "la source produit un signal d'activation qui active la ou les destinations". Il se différencie d'un arc par le fait qu'un sommet peut avoir plusieurs successeurs directs. Chaque sommet sans successeur est un bloc standard et tout autre sommet correspond à un bloc *if then else* possédant un seul prédécesseur et au moins un successeur. Ces sommets sont la source d'au plus deux hyperarcs, un pour la branche *then* et un pour la branche *else*. Chaque hyperarc peut avoir pour destination au plus un bloc *if then else*, mais un nombre illimité de blocs standards. Les blocs standards pouvant être multi-activés, ils peuvent avoir plusieurs prédécesseurs, ce qui différencie un HOAA d'un arbre.

La figure 3.32 page 91 montre à gauche un schéma-bloc Scicos issu de la phase de pré-compilation et à droite le HOAA correspondant. À chaque bloc correspond un numéro. Il apparaît dans le HOAA que les blocs 5, 6 et 8 qui correspondent aux entrées et sorties du *superbloc* sont activés par le signal d'activation d'entrée du *superbloc*. Celui-ci active aussi le bloc *if then else* 1 qui active à son tour le bloc *if then else* 2 sur sa branche *then* et les blocs *sample and hold* 3 et *if then else* 9 sur sa branche *else*. Enfin le bloc *sample and hold* 4 est activé à la fois par la branche *else* du bloc *if then else* 2 et la branche *else* du bloc *if then else* 9.

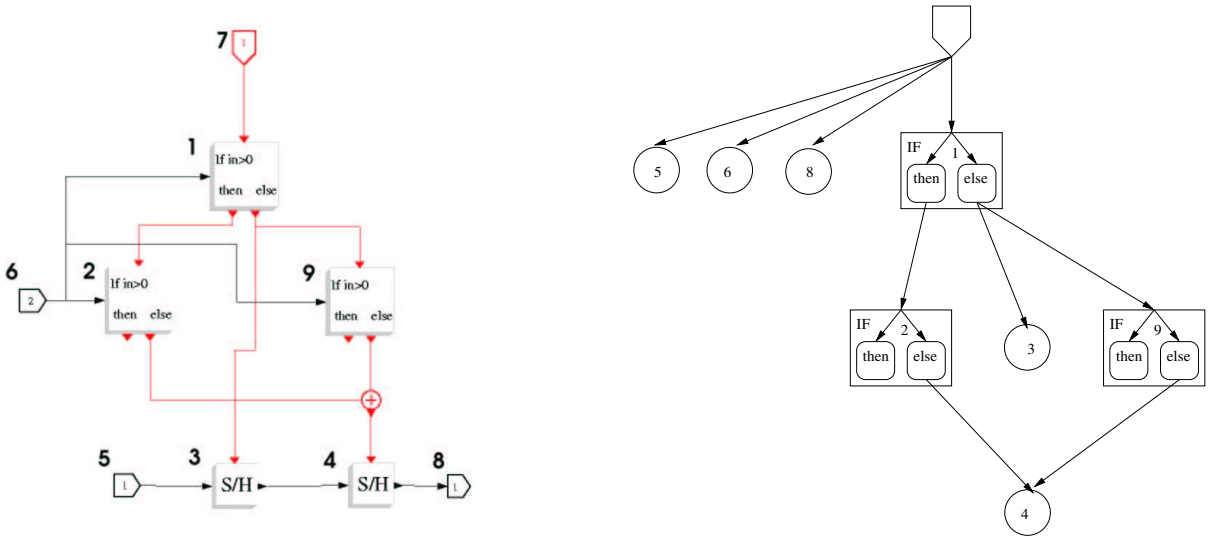

FIG. 3.32 – *Schéma-bloc Scicos et HOAA correspondant*

Il est possible ensuite d'exploiter ce HOAA pour connaître la relation qui existe entre les activations d'un bloc producteur et celles d'un bloc consommateur. Si l'on omet d'abord les cas de multi-activation, il est possible de reconnaître chaque relation :

– $A_P = A_C$: si les deux blocs sont les destinations d'un seul et même hyperarc ;

- $A_C \subset A_P$: si dans le chemin permettant d'aller du sommet source de l'HOAA au bloc Y, il existe un hyperarc qui a pour destination le bloc X mais pas le bloc Y ;
- $A_P \subset A_C$: si dans le chemin permettant d'aller du sommet source de l'HOAA au bloc X, il existe un hyperarc qui a pour destination le bloc Y mais pas le bloc X ;
- $A_P \cap A_C = \emptyset$: si la chaîne minimale (en nombre d'hyperarcs empruntés) qui permet d'aller du bloc X au bloc Y emprunte les deux hyperarcs *then* et *else* d'un même bloc *if then else*.

En considérant la multi-activation, il faut vérifier que la relation est vraie pour chacun des différents signaux d'activation, ce qui conduit à reconnaître chaque relation de manière légèrement différente qu'énoncée précédemment :

- $A_P = A_C$: si un hyperarc possède pour destination P, il possède également pour destination C et inversement ;
- $A_C \subset A_P$: si pour tout chemin permettant d'aller du sommet source de l'HOAA au bloc C, il existe un hyperarc qui a pour destination le bloc P et que, soit pour au moins un de ces chemins l'hyperarc qui a pour destination P n'a pas pour destination C, soit P est la destination d'au moins un hyperarc qui n'a pas été parcouru ;
- $A_P \subset A_C$: si pour tout chemin permettant d'aller du sommet source de l'HOAA au bloc P, il existe un hyperarc qui a pour destination le bloc C et que, soit pour au moins un de ces chemins l'hyperarc qui a pour destination C n'a pas pour destination P, soit C est la destination d'au moins un hyperarc qui n'a pas été parcouru ;
- $A_P \cap A_C = \emptyset$: si, pour chaque hyperarc ayant pour destination P ou C, la chaîne minimale le contenant et permettant d'aller du bloc P au bloc C emprunte les deux hyperarcs *then* et *else* d'un même bloc *if then else*;
- *Cas mixte* : si aucun des cas précédents n'est vérifié.

Ces méthodes de détection annulent et remplacent les précédentes. Si on revient au HOAA de la figure 3.32 page 91, l'arc régulier entre le bloc 5 et le bloc 3 correspond au $2^{ème}$ cas ($A_C \subset A_P$) car dans le chemin menant du sommet source (numéro 7) jusqu'au bloc 3, le premier hyperarc a également pour destination le bloc 5. Au contraire, l'arc régulier entre le bloc 4 et le bloc 8 correspond au $3^{ème}$ cas ($A_P \subset A_C$) car dans les deux chemins menant du sommet source (numéro 7) jusqu'au bloc 4, il existe un hyperarc (le premier) qui a également pour destination le bloc 8. L'arc régulier du bloc 3 au bloc 4 correspond quant à lui au dernier cas. Néanmoins on peut observer que si on ne considère que l'activation du bloc 4 par la branche *else* du bloc *if then else*2, l'arc régulier entre les blocs 3 et 4 correspond au $4^{ème}$ cas ($A_P \cap A_C = \emptyset$). En effet la chaîne minimale allant du bloc 3 au bloc 4 passe par le bloc *if then else*1 en utilisant ses deux hyperarcs *then* et *else*.

A chaque sommet *if then else* d'un HOAA est associé un "niveau". S'il existe, le sommet *if then else* successeur direct du sommet source est de niveau 1. Ces successeurs directs *if then else* sont de niveau 2, leurs successeurs directs de niveau 3 et ainsi de suite. Le HOAA permettra ainsi de déterminer l'Ancêtre d'Activation Commune le Plus Proche (AACPP) de deux blocs P et C. Il s'agit du bloc *if then else* de niveau le plus élevé tel que si on considère le graphe constitué de ces successeurs [15] et des arcs les reliant, ce graphe contient tous les hyperarcs de l'HOAA ayant pour

15. directs et indirects

destination les deux blocs. Dans la traduction, cela veut dire qu'une opération conditionnante qui contient toutes les instances des blocs P et C dans ses sous-graphes [16] est l'opération conditionnante correspondant à l'AACPP de deux blocs P et C ou contient cette opération. Cette information sera utile dans les cas de rémanence.

Nous allons maintenant détailler la traduction d'un arc régulier dans chaque cas de relation entre les signaux d'activation du producteur et du consommateur. Nous rappellerons à chaque fois ce que signifie cette relation, nous montrerons comment elle influence la traduction de l'arc régulier dans le cas où ni le consommateur ni le producteur sont multi-activés, avant d'élargir au cas multi-activé. Nous rappelons encore une fois que dans le cas de multi-activation, tous les signaux d'activation sont exclusifs deux à deux.

3.3.3.6.1 Cas $A_C \subset A_P$

La relation entre les signaux d'activation du producteur et ceux du consommateur indique ici que, lorsque le consommateur est activé, le producteur l'est aussi. Toute donnée consommée a donc forcément été produite dans le même instant logique : il n'y a donc pas besoin de rémanence. L'arc régulier est donc traduit en dépendance de données pour forcer la précédence du producteur sur le consommateur.

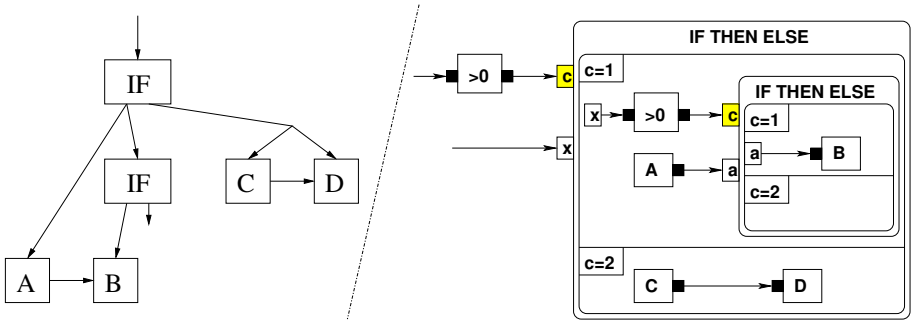

FIG. 3.33 – *Cas* $A_C \subset A_P$

A gauche de la figure 3.33 page 93 est représenté un HOAA d'un schéma-bloc Scicos possédant deux bloc *if then else* et 4 blocs standards A, B, C et D. Puisqu'un bloc *if then else* active l'autre, cela se traduit par la présence d'une opération conditionnante *if then else* dans l'un des sous-graphes d'une autre opération conditionnante *if then else*, comme le montre le graphe SynDEx représenté à droite de la même figure. Comme expliqué au 3.3.3.3 page 86, chaque bloc est traduit par une opération dans le sous-graphe de l'opération conditionnante correspondant à la branche du bloc *if then else* qui l'active :

- A dans le sous-graphe du cas $c = 1$ (branche *then*) de l'opération conditionnante de plus haut niveau,
- B dans le sous-graphe du cas $c = 1$ (branche *then*) de l'opération conditionnante contenue dans le sous-graphe correspondant au cas $c = 1$ (branche *then*) de l'opération conditionnante de plus haut niveau,

16. ce qui comprend tous les niveaux de hiérarchie inférieurs

– C et D dans le sous-graphe du cas $c = 2$ (branche *else*) de l'opération conditionnante de plus haut niveau,

L'arc régulier tel que $A_C \subset A_P$ est celui reliant le bloc A au bloc B. Les opérations correspondant à ces deux blocs n'étant pas au même niveau de la hiérarchie, un dépendance de données doit relier le producteur (ici A) à l'opération conditionnante du même niveau que lui et contenant le consommateur. Cette dépendance est ensuite propagée dans la hiérarchie (via plusieurs niveaux de hiérarchie si nécessaire) jusqu'au consommateur. C'est ce qui est visible sur le graphe SynDEx de droite de la figure 3.33 page 93. L'arc régulier entre les blocs C et D est quant à lui traduit par une dépendance de données entre deux opérations de même niveau puisque $A_C = A_D$.

Si $A_C \subset A_P$ et qu'il y a multi-activation du consommateur seulement (par exemple ici si le bloc B était également activé par la branche *else* du bloc *if then else* de niveau 2), cela ne change rien si ce n'est que le consommateur est traduit par plusieurs instances (une pour chaque signal d'activation) d'une même opération placées dans des sous-graphes différents. En effet la relation implique que le bloc producteur sera de toute façon à un niveau hiérarchique supérieur à ces instances : il suffira seulement de propager la dépendance de données vers les différentes instances du consommateur (par exemple il y aurait ici une autre instance de B dans l'autre sous-graphe de l'opération conditionnante de niveau hiérarchique inférieur).

Si $A_C \subset A_P$ et qu'il y a multi-activation du producteur seulement, cela signifie que le signal d'activation du consommateur est un sous-échantillonnage d'un des signaux d'activation du producteur mais est exclusif avec les autres. Par exemple, si on reprend l'HOAA de la figure 3.33 page 93 et que A est aussi activé par la branche *else* du bloc *if then else* de niveau 1, le signal d'activation de B est un sous-échantillonnage de l'activation par la branche *then* mais est exclusif avec le signal d'activation de la branche *else*. Du coup, bien que le bloc A soit traduit par deux instances d'une même opération, ce sera obligatoirement celle de la branche *then* qui sera exécutée lorsque l'opération correspondant au bloc B le sera. Il n'est donc pas possible qu'une donnée produite par l'autre instance de A soit consommée par B. N'est donc considéré pour la traduction de l'arc régulier que le signal d'activation du producteur dont le signal d'activation du consommateur est un sous-échantillonnage. Aucune dépendance n'est donc à ajouter par rapport au cas sans multi-activation.

Si $A_C \subset A_P$ et qu'il y a à la fois multi-activation du producteur et du consommateur, il suffit de considérer chaque signal d'activation du consommateur. Il existe alors parmi les signaux d'activation du producteur un seul signal d'activation qui ne soit pas exclusif avec celui du consommateur. On traite alors le problème pour ces deux signaux comme indiqué dans le cas sans multi-activation.

3.3.3.6.2 Cas $A_P \cap A_C = \emptyset$

La relation entre les signaux d'activation du producteur et ceux du consommateur indique ici qu'il ne peut y avoir activation et donc exécution du producteur et du consommateur lors du même instant logique. Cela signifie que la donnée produite lors d'un instant logique n'est pas consommée lors de ce même instant logique et doit donc être rendue disponible lors de l'instant logique suivant. Il conviendra donc ici de traduire l'arc régulier par un sommet *retard* et des dépendances de données connectant celui-ci au producteur et au consommateur.

C'est ce que montre la figure 3.34 page 95 avec le graphe SynDEx à droite correspondant à

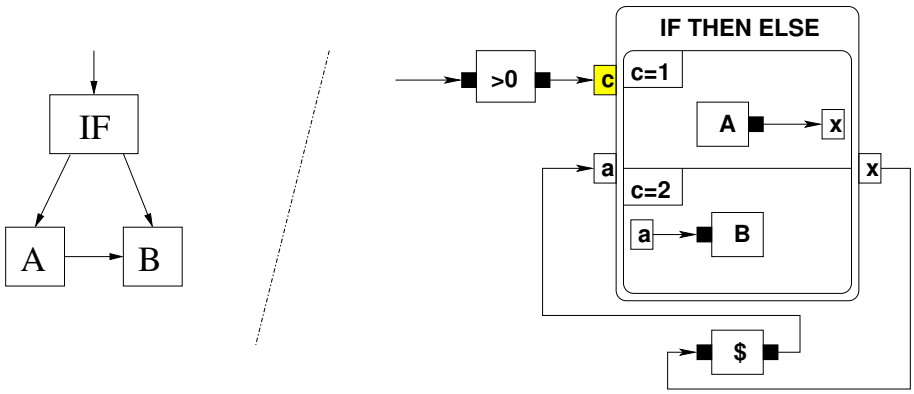

FIG. 3.34 – *Cas $A_P \cap A_C = \emptyset$, traduction erronée*

l'HOAA de gauche. Le producteur (sommet *A*) et le consommateur (sommet *B*) sont dans deux sous-graphes différents de l'opération conditionnante *if then else*. Le producteur est relié à une opération *retard* via le port de sortie *x* de l'opération conditionnante et cette opération *retard* est connectée au consommateur via le port *a*. Une donnée produite par l'opération *A* peut ainsi être consommée par l'opération *B* lors de l'instant logique suivant. Mais si le sous-graphe auquel appartient l'opération *B* est exécuté lors de deux instants logiques consécutifs, la valeur de la donnée consommée par l'opération *B* lors du deuxième instant logique est indéterminée : l'opération A n'ayant pas été exécutée lors de l'instant logique précédent, l'opération *retard* ne dispose d'aucune donnée précédente à produire. Il faut donc fournir à cette opération *retard* une donnée à consommer même lorsque l'opération *A* n'est pas exécutée [17]. Il suffit pour cela, dans le sous-graphe correspondant à la branche *else* ($c = 2$), de faire consommer à l'opération *retard* la donnée qu'il a produit en connectant dans ce sous-graphe le port d'entrée *a* au port de sortie *x* comme le montre la figure 3.35 page 96.

Ainsi, il ne suffit pas d'intercaler une opération *retard* entre le producteur et le consommateur il faut également que, pour tous les sous-graphes dans lesquels le producteur ne possède pas d'instance, la sortie de cette opération *retard* soit connectée à son entrée. C'est ce que montre la figure 3.36 page 96 avec un cas de HOAA comprenant deux blocs *if then else*. La traduction comporte alors deux opérations conditionnantes *if then else* dans les sous-graphes desquels, soit il existe une opération *A*, soit l'entrée correspondant à l'opération retard est connectée à la sortie correspondant également à cette opération.

Il reste à savoir à quel niveau de hiérarchie doit être placée cette opération *retard*. Dans la hiérarchie du graphe SynDEx, il doit être placé assez haut pour que sa sortie puisse être consommée par toutes les instances du consommateur et que son entrée puisse consommer la donnée de n'importe quelle instance du producteur. C'est pourquoi le sommet *retard* est placé au même niveau que l'opération conditionnante *if then else* correspondant à l'AACPP du bloc producteur et du bloc consommateur.

17. On aurait de toute façon remarqué que le graphe SynDEx de droite de la figure 3.34 page 95 n'est pas correct, un port de sortie d'une opération conditionnante devant être connecté dans tous les sous-graphes.

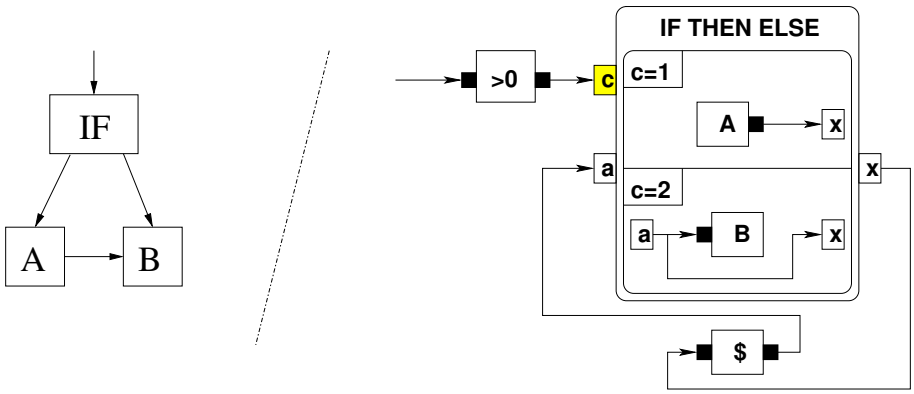

FIG. 3.35 – *Cas $A_P \cap A_C = \emptyset$, traduction correcte*

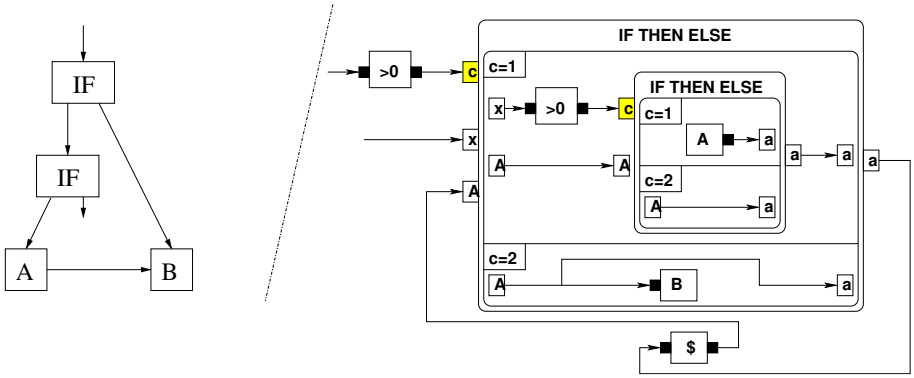

FIG. 3.36 – *Cas $A_P \cap A_C = \emptyset$ hiérarchique, traduction correcte*

3.3.3.6.3 Cas $A_P \subset A_C$

La relation entre les signaux d'activation du producteur et ceux du consommateur indique ici que l'exécution du producteur est toujours accompagnée de celle du consommateur, mais que ce dernier peut aussi être activé et exécuté seul. Toute donnée produite est donc consommée lors du même instant logique. Lorsque le consommateur est exécuté seul, il y a donc consommation d'une donnée produite lors d'un instant logique précédent et ayant déjà été consommée.

Le signal d'activation du producteur étant un sous-échantillonnage de celui du consommateur, dans le graphe SynDEx l'opération correspondant au producteur se trouve dans un des sous-graphes d'une opération conditionnante *if then else* du même niveau hiérarchique que l'opération correspondant au consommateur. Le cas où les deux opérations sont exécutées se traduit donc par une dépendance de données allant du producteur au consommateur via la hiérarchie. C'est ce que montre la figure 3.37 page 97.

Mais cette première traduction est incomplète car elle ne prend pas en compte le cas où le consommateur est exécuté seul : par exemple sur la figure 3.37 page 97, si $c = 1$ dans l'opération conditionnante *if then else* de plus haut niveau et $c = 2$ pour l'opération conditionnante de plus

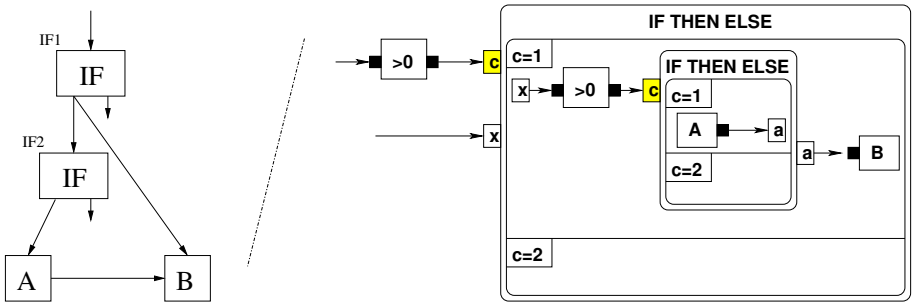

FIG. 3.37 – *Cas $A_P \subset A_C$, traduction erronée*

bas niveau. Il faut alors utiliser une opération *retard*. Celle-ci est placée au même niveau hiérarchique que l'opération *if then else* correspondant à l'AACPP du consommateur et du producteur, c'est-à-dire, dans le cas sans multi-activation, le bloc *if then else* activant le consommateur par l'une de ses branches. Cette opération *retard* consomme la donnée produite par le producteur ou, dans le cas où il n'est pas exécuté, la donnée qu'elle a elle-même (l'opération *retard*) produite. Cela implique que pour l'opération *if then else* correspondant à l'AACPP et dans les opérations *if then else* qu'elle contient (en considérant tous les niveaux de hiérarchies), si un des sous-graphes possède une instance du producteur :

- cette opération *if then else* possède un port d'entrée connecté (via la hiérarchie) au port de sortie de l'opération *retard*,
- cette opération *if then else* possède un port de sortie connecté (via la hiérarchie) au port d'entrée de l'opération *retard*,
- dans le sous-graphe qui contient le producteur, une dépendance de données existe entre lui et le port de sortie connecté à l'opération *retard*,
- dans le sous-graphe qui ne contient pas le producteur, cette entrée et cette sortie sont connectées ensemble par une dépendance de données.

Une traduction correcte d'un arc régulier lorsque le producteur est activé par un sous-échantillonnage du signal d'activation du consommateur est donnée par la figure 3.38 page 98.

Si $A_P \subset A_C$ et qu'il y a multi-activation du consommateur seulement (par exemple ici si le bloc *B* était également activé par la branche *else* du bloc *if then else* de niveau 1), cela signifie que le signal d'activation du producteur est un sous-échantillonnage d'un des signaux d'activation du consommateur, mais est exclusif avec les autres. Les instances du consommateur correspondant à ces signaux d'activation exclusifs sont donc dans des sous-graphes ne contenant pas d'instances du producteur ni à ce niveau de hiérarchie ni dans les niveaux inférieurs. Ces instances consomment donc la donnée produite par l'opération *retard*, qui se trouve forcément dans un niveau hiérarchique supérieur grâce au calcul de l'AACPP.

Si $A_P \subset A_C$ et qu'il y a multi-activation du producteur seulement, cela ne pose pas de problème si ce n'est qu'il existe plusieurs instances du producteur appartenant à des sous-graphes différents [18] de l'opération conditionnante *if then else* placée dans le même sous-graphe que le

18. à ce niveau de hiérarchie ou bien dans les niveaux inférieurs.

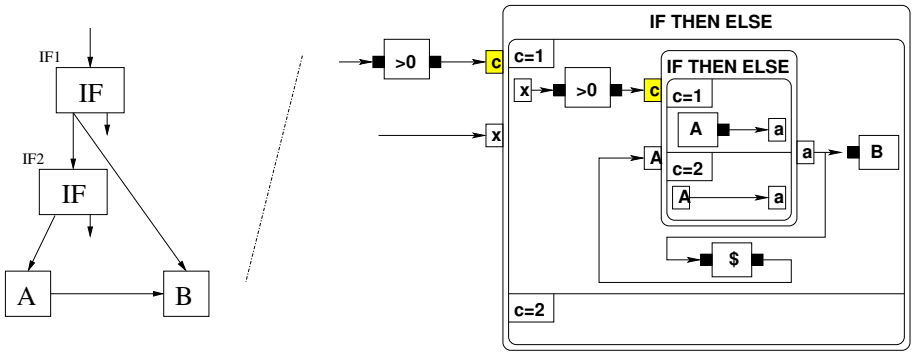

FIG. 3.38 – *Cas $A_P \subset A_C$, traduction correcte*

consommateur. Les règles précédentes s'appliquent.

Si $A_P \subset A_C$ et qu'il y a à la fois multi-activation du producteur et du consommateur, il suffit de considérer chaque signal d'activation du producteur. Il existe alors parmi les signaux d'activation du consommateur un seul signal d'activation qui ne soit pas exclusif avec celui du producteur. On traite alors le problème pour ces deux signaux, comme dans le cas sans multi-activation. Pour les instances du consommateur correspondant à des signaux d'activation exclusifs avec ceux du producteur, on utilise la règle énoncée pour le cas de multi-activation du consommateur seul.

3.3.3.6.4 Cas mixte

La relation entre les signaux d'activation du producteur et ceux du consommateur indique ici que le producteur peut être activé seul, le consommateur aussi, ou ils peuvent être activés ensemble. Cette relation n'est possible que s'il y a multi-activation d'au moins un des deux blocs. La figure 3.39 page 99 présente un exemple de schéma-bloc comportant deux blocs standards multi-activés et dont les signaux d'activation correspondent au cas mixte.

Parce que producteur et consommateur peuvent être activés séparément, il y a rémanence et donc, dans la traduction, une opération *retard*, placée au même niveau que l'opération conditionnante correspondant au bloc qui est l'AACPP du consommateur et du producteur. Sur l'exemple de la figure 3.39 page 99, l'AACPP des blocs A et B est le bloc *if then else* noté $IF0$.

Une fois cette opération *retard* placée, on commence par identifier parmi les signaux d'activation du consommateur et du producteur ceux qui sont exclusifs avec tous les autres. On applique alors pour les instances correspondantes les règles de traduction des arcs réguliers du cas $A_P \cap A_C = \emptyset$. Toujours dans notre exemple de le figure 3.39 page 99, cela correspond à ne considérer que le signal s_4 pour le bloc A et s_5 pour le bloc B, en effet $s_4\ X_{cl}\ s_5$.

Considérons à présent les ensembles des signaux d'activation restant que l'on notera A'_P et A'_C (le schéma-bloc (A) de la figure 3.40 page 100 applique ce principe au schéma-bloc de la figure 3.39 page 99). Pour chacun de ceux activant le producteur (respectivement le consommateur), il existe un signal d'activation activant le consommateur (respectivement le producteur) tel que ces deux signaux ne soient pas en exclusion. Un signal d'activation du producteur (respectivement du consommateur) peut ne pas être en exclusion avec plusieurs signaux d'activation du consommateur

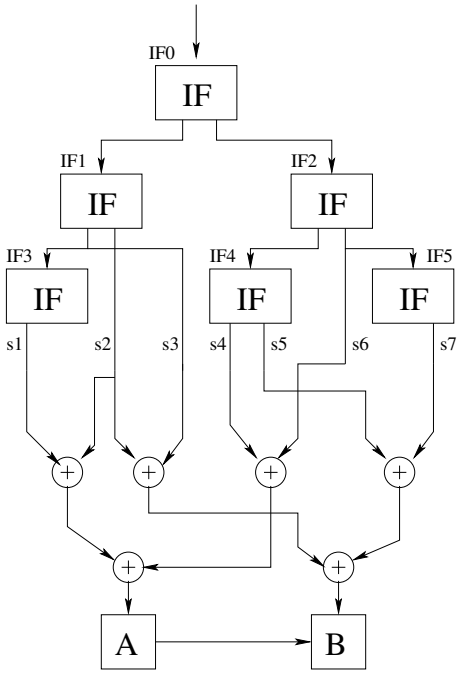

FIG. 3.39 – *Schéma-bloc correspondant au cas mixte*

(respectivement du producteur). Par cette relation, on construit ainsi des ensembles de signaux d'activation de trois types :

- contenant un signal d'activation du producteur et un du consommateur, et ces deux signaux sont en réalité le même signal. On applique alors les règles du cas $A_P = A_C$. C'est le cas avec le signal s_2 du schéma-bloc de la figure 3.39 page 99, comme le montre le schéma-bloc (B) de la figure 3.40 page 100 ;
- contenant un signal d'activation du producteur et un ou plusieurs signaux d'activation du consommateur, différents de celui du producteur. On applique alors les règles du cas $A_C \subset A_P$. C'est le cas avec les signaux s_6 et s_7 du schéma-bloc de la figure 3.39 page 99, comme le montre le schéma-bloc (C) de la figure 3.40 page 100 ;
- contenant un signal d'activation du consommateur et un ou plusieurs signaux d'activation du producteur, différents de celui du producteur. On applique alors les règles du cas $A_P \subset A_C$. C'est le cas avec les signaux s_1 et s_3 du schéma-bloc de la figure 3.39 page 99, comme le montre le schéma-bloc (D) de la figure 3.40 page 100.

Dans tous les cas, une dépendance de données doit connecter chaque instance du producteur à l'opération *retard* (via la hiérarchie, bien sûr), même si cette instance est déjà connectée à une instance du consommateur.

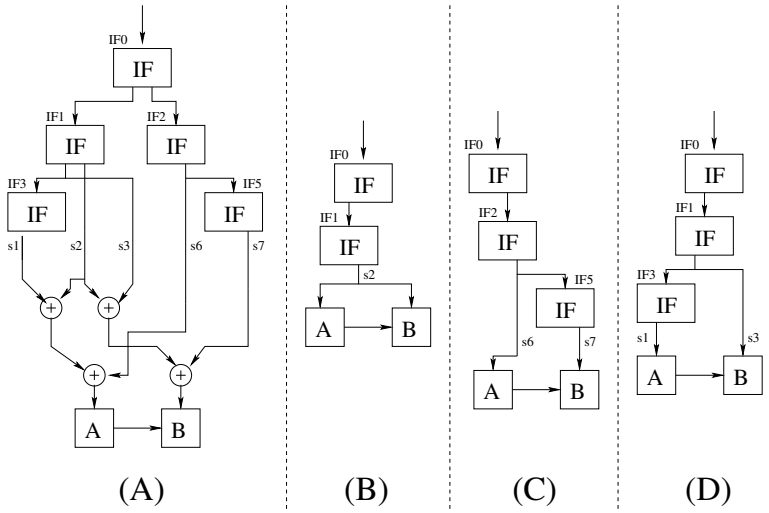

FIG. 3.40 – *Différentes étapes de traduction du schéma-bloc de la figure 3.39 page 99*

3.3.3.7 Exemple

La figure 3.41 page 101 montre le graphe SynDEx résultant de la traduction du schéma-bloc Scicos de la figure 3.32 page 91. Le bloc producteur 3 n'est activé que par un seul signal d'activation mais le bloc consommateur 4 est activé par deux signaux d'activation : l'un est en exclusion avec celui du producteur, l'autre en est un sous-échantillonnage. C'est donc le cas $A_C \subset A_P$ avec multi-activation du consommateur qui est appliqué.

3.3.3.8 Conclusion

Nous avons ici énoncé les principes de traduction en programmes SynDEx des schémas-blocs Scicos, notamment le conditionnement et la rémanence. Cette traduction a fait l'objet d'une implémentation complète dans le compilateur Scicos et a été testée sur une application industrielle significative comme nous le verrons au chapitre 5. Ces principes de traduction peuvent être facilement adaptés à d'autres langages utilisant le modèle schéma-bloc, comme Simulink. Cela permet ainsi de passer d'un langage adapté à la simulation (Scicos, Simulink) à un langage adapté à l'implantation distribuée, SynDEx. Il nous semblait important de ne pas se limiter à la traduction de langage flot de contrôle, le conditionnement pouvant aussi être exprimé par des langages flot de données. C'est la problématique de la rémanence qui nous a conduit à présenter ici la traduction d'un langage schéma-bloc plutôt que d'un langage flot de données "classique" permettant le conditionnement, comme les langages IDF, Lustre ou Signal. Ce dernier a d'ailleurs lui aussi fait l'objet d'une traduction [65] en langage SynDEx dans le cadre du projet ACOTRIS.

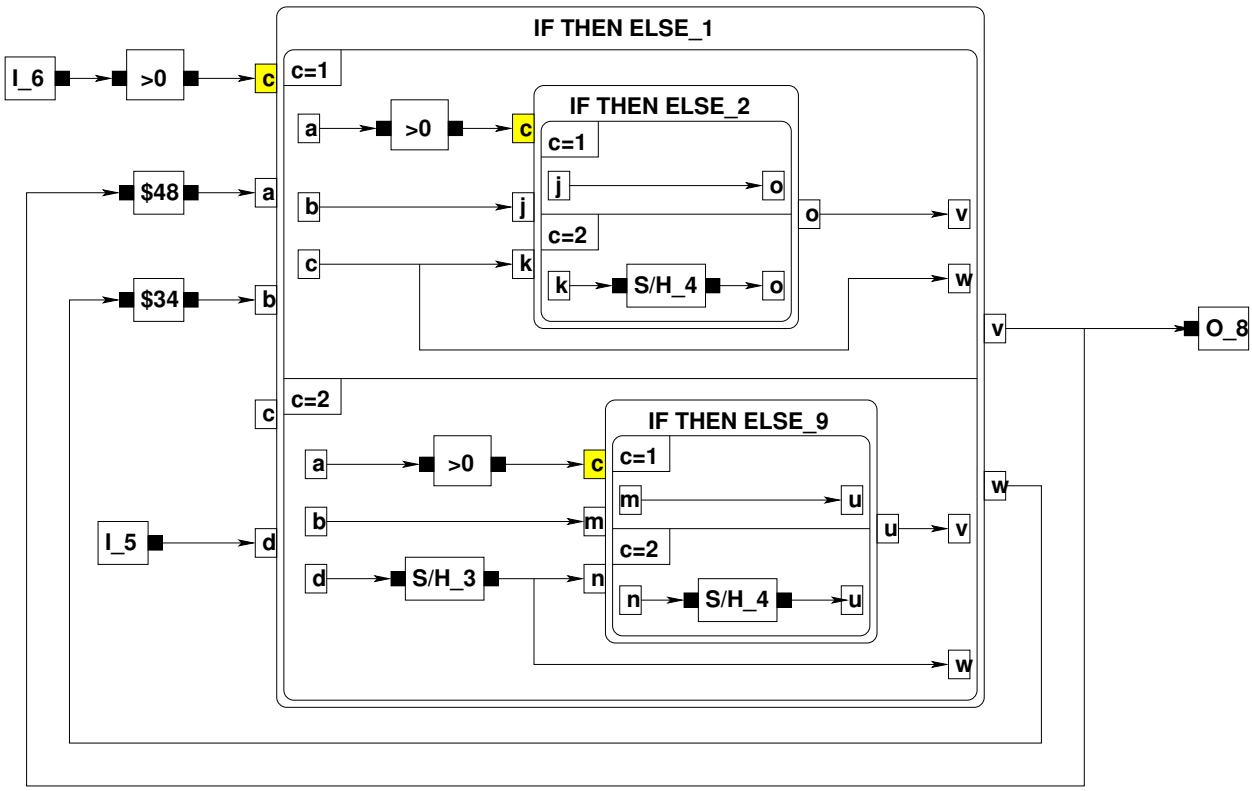

FIG. 3.41 – *Graphe SynDEx obtenu par traduction du schéma-bloc Scicos de la figure 3.32 page 91*

Chapitre 4

Implantation distribuée avec le logiciel SynDEx

4.1 Principes du logiciel SynDEx

SynDEx* [1] [66] est un logiciel libre de CAO niveau système développé par le projet AOSTE de l'INRIA Rocquencourt, projet dans lequel s'est déroulé cette thèse. Ce logiciel repose sur la méthodologie Adéquation Algorithme Architecture (AAA) [35] qui consiste à réaliser l'adéquation, c'est-à-dire l'implantation optimisée, d'un algorithme sur une architecture. Il s'agit alors d'exploiter le parallélisme potentiel de l'algorithme pour utiliser au mieux le parallélisme physique de l'architecture. Ce type de problème étant d'une complexité combinatoire exponentielle, on cherche à obtenir dans un temps raisonnable une solution approchée. On utilise donc pour la distribution et l'ordonnancement une heuristique qui tend à minimiser le temps d'exécution total de l'algorithme [67]. Cette heuristique gloutonne d'ordonnancement par liste [68] prend en compte le coût et l'ordonnancement des communications. La méthodologie AAA utilise des graphes pour décrire l'algorithme et l'architecture et ce sont des transformations de graphes qui permettent d'obtenir les implantations distribuées possibles. Le langage utilisé pour décrire l'algorithme est le langage SynDEx reposant sur le modèle flot de données conditionné précédemment décrit (voir chapitre 2).

Les principes du logiciel sont illustrés par la figure 4.1 page 104. L'utilisateur décrit à l'aide du logiciel SynDEx un algorithme et une architecture. L'algorithme est un sous-graphe motif (infiniment répété) utilisant le langage SynDEx. Il est également possible d'y décrire de la répétition finie, un seul sommet pouvant par exemple correspondre à un traitement effectué sur chaque scalaire du vecteur qu'il consomme en entrée [69]. La figure 4.2 page 105 montre un exemple d'algorithme décrit à l'aide du logiciel SynDEx. Ce graphe comporte 3 capteurs, c'est-à-dire des opérations sans prédécesseur, colorés en rouge pâle et 3 actionneurs, c'est-à-dire des opérations sans successeur, colorés en rouge foncé. Il comporte également 11 autres opérations (en bleu) réalisant des traitements entre les entrées acquises par les capteurs et celles retournées par les actionneurs.

L'architecture est quant à elle décrite à l'aide d'un graphe non orienté où chaque nœud est un

1. http://www.syndex.org

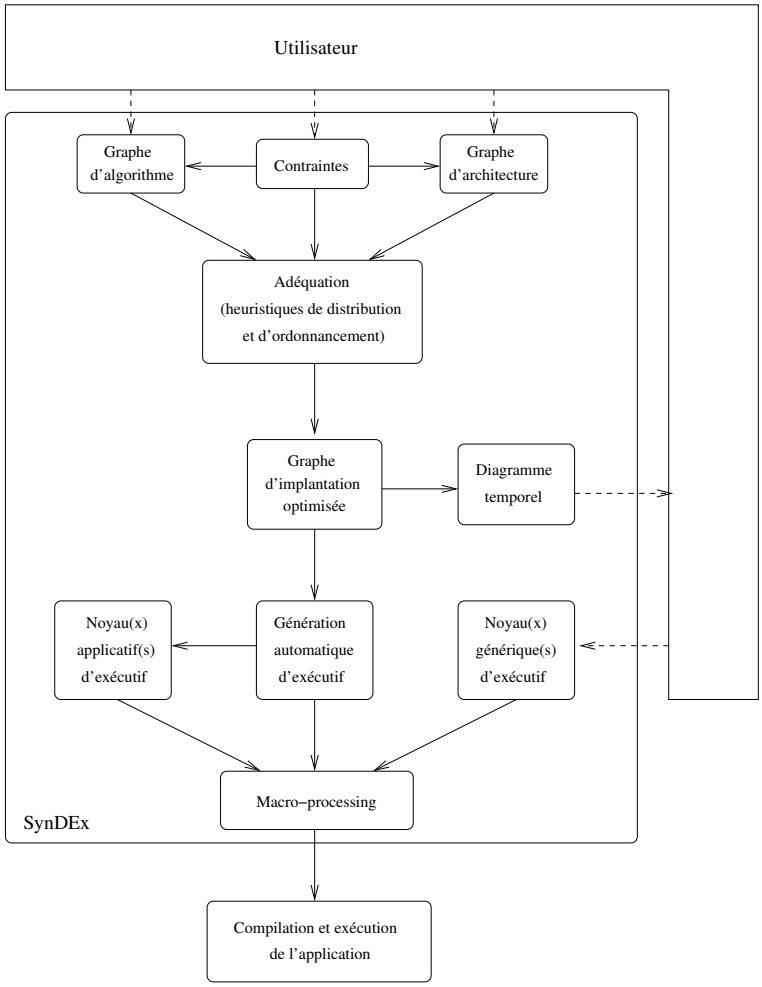

FIG. 4.1 – *Principes du logiciel SynDEx*

processeur ou un média de communication et où chaque arc est une connexion entre un processeur et un média [70]. Un média de communication peut être une liaison point à point, une liaison multi-point (bus) avec ou sans broadcast, ou une mémoire partagée. Ce modèle de représentation des architectures distribuées est une extension des modèles classiquement utilisés pour spécifier les architectures parallèles ou distribuées que sont les PRAM ("Parallel Random Access Machines") et les DRAM ("Distributed Random Access Machines") [71]. Il permet de décrire des architectures utilisant les deux types de communication que sont les mémoires partagées (PRAM) et le passage de message (DRAM).

Des extensions de ce modèle permettant de raffiner la description de l'architecture, par exemple en décrivant les différentes unités de calcul (ALU,FPU etc ...) et interfaces de communication (DMA, convertisseur série/parallèle etc ...) d'une carte DSP, sont proposées dans [70] et [72].

La figure 4.3 page 105 montre un exemple d'architecture décrite à l'aide du logiciel SynDEx. Celle-ci comporte 3 processeurs de type MPC555 (PowerPC de Motorola) connectés par un média

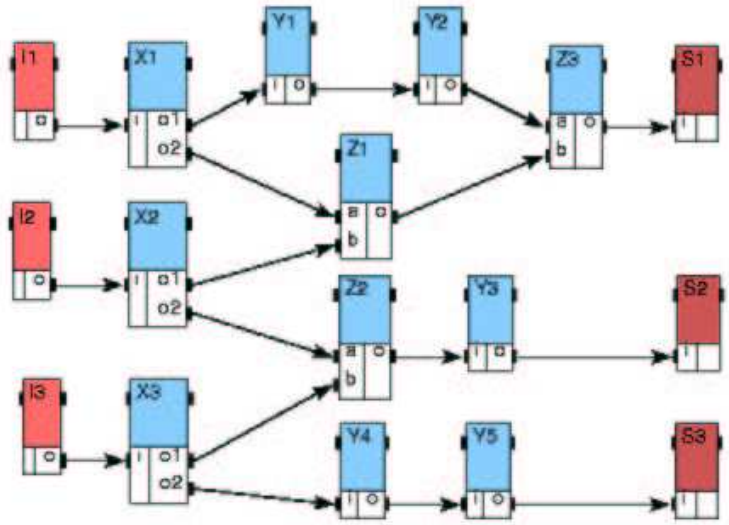

FIG. 4.2 – *Exemple d'algorithme décrit avec le logiciel SynDEx*

de communication de type CAN (Controller Area Network).

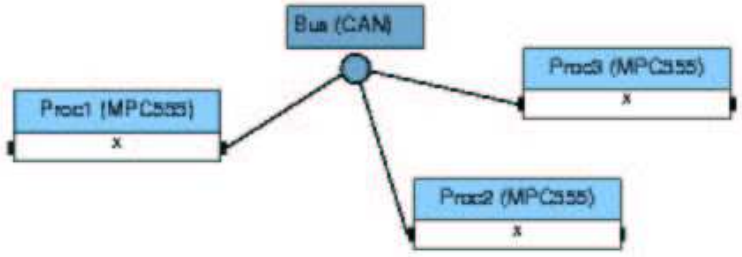

FIG. 4.3 – *Exemple d'architecture décrite avec le logiciel SynDEx*

Une fois l'algorithme et l'architecture décrits, l'utilisateur doit préciser les contraintes. Ces contraintes incluent la caractérisation des opérations et des dépendances de données relativement à l'architecture. Ainsi pour chaque opération du graphe d'algorithme, l'utilisateur doit indiquer un temps d'exécution au pire cas (WCET pour "Worst Case Execution Time") sur chaque type de processeur de l'architecture. De même, il doit être indiqué pour chaque type de dépendance de données (entier, image, etc ...) un temps de transfert au pire cas sur chaque type de média de communication de l'architecture. Des contraintes de distribution peuvent également être spécifiées, obligeant telle opération à s'exécuter sur tel processeur ou tel groupe de processeurs.

Lorsque ces contraintes sont spécifiées, l'utilisateur peut demander au logiciel d'effectuer l'adéquation. Celle-ci utilise une heuristique gloutonne qui prend en compte les estimations de temps d'exécution et de transfert pour ordonnancer et distribuer l'algorithme sur l'architecture en effectuant des transformations de graphe [67]. Sont notamment rajoutés des opérations pour gérer

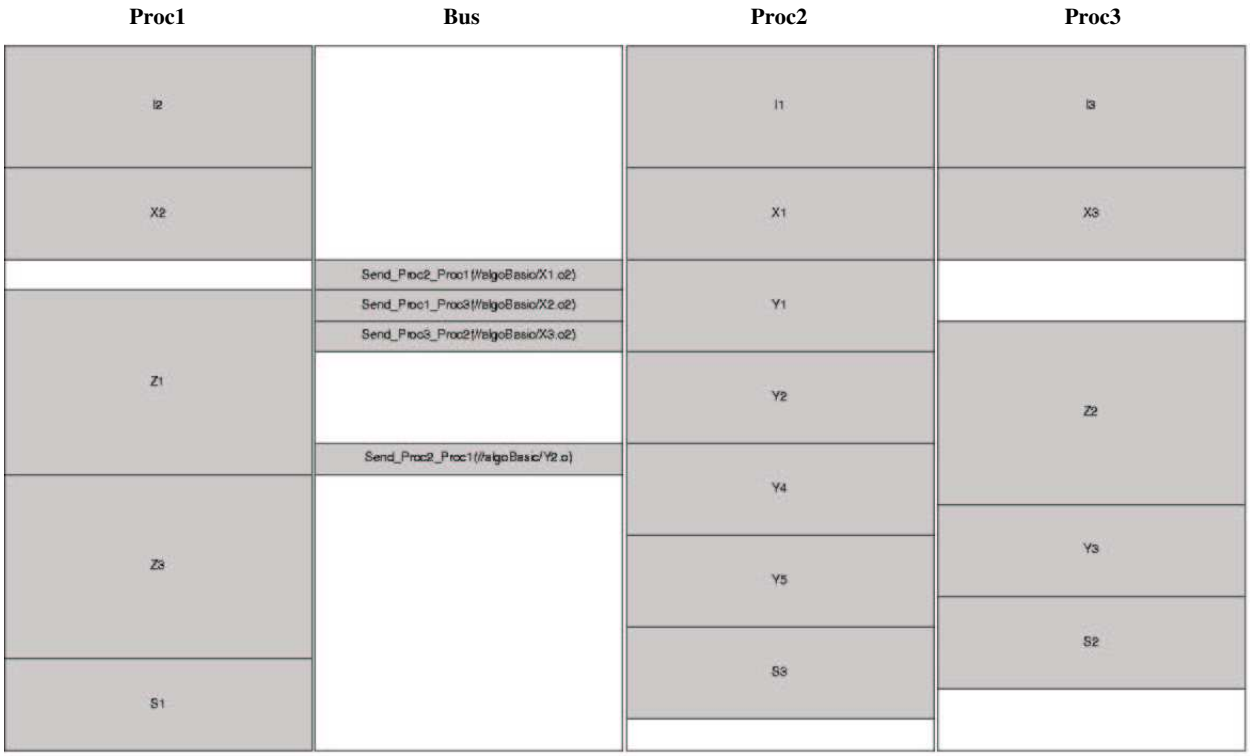

FIG. 4.4 – *Diagramme temporel résultant de l'adéquation de l'algorithme de la figure 4.2 avec l'architecture de la figure 4.3*

les envois, les réceptions et les synchronisations entre les processeurs [73], afin que l'ordre partiel obtenu après ordonnancement et distribution soit compatible avec l'ordre partiel du graphe d'algorithme initial. Une fois l'adéquation réalisée, l'utilisateur peut visualiser un diagramme temporel représentant la distribution et l'ordonnancement de l'algorithme sur l'architecture. La figure 4.4 page 106 montre un exemple de diagramme temporel que peut voir l'utilisateur après adéquation de l'algorithme de la figure 4.2 page 105 sur l'architecture 4.3 page 105 à l'aide du logiciel SynDEx. Ce diagramme temporel se lit de haut en bas. A chaque colonne correspond un processeur ou un média de communication. Sur la figure 4.4, on a de gauche à droite le processeur *Proc*1, le bus CAN, le processeur *Proc*2 et enfin le processeur *Proc*3. On s'aperçoit que l'adéquation a permis d'utiliser pleinement le parallélisme physique de l'architecture pour distribuer les opérations équitablement sur les 3 processeurs. Si l'ordonnancement et la distribution qu'illustre le diagramme temporel ne satisfait pas l'utilisateur, par exemple parce que les contraintes temps réel ne sont pas satisfaites, l'utilisateur peut réaliser une nouvelle adéquation après avoir modifié au choix :

- le graphe d'algorithme pour augmenter le parallélisme potentiel ou la granularité,
- le graphe d'architecture en augmentant le nombre de processeurs ou en diminuant celui-ci si au contraire les contraintes temps réel sont satisfaites. Cela permet ainsi de faire de l'exploration d'architecture ;
- les contraintes de distribution afin d'éviter les résultats aberrants, inhérents à toutes les heu-

ristiques.

Une fois l'utilisateur satisfait par l'une des adéquations, il peut demander au logiciel SynDEx de générer le code correspondant à celle-ci. Il s'agit en réalité de fichiers contenant du macro-code M4 [74]. Un fichier de ce type est généré pour chaque processeur de l'architecture. Chaque fichier contient à la fois des macros correspondant aux opérations de l'algorithme, mais aussi des macros de "service bas-niveau" qui permettent à terme une exécution sans système d'exploitation résident : synchronisations, primitives d'envoi et de réception de données, gestion des tâches, etc. Ce macro-code généré par le logiciel SynDEx nécessite d'être ensuite macro-processé pour obtenir des fichiers compilables puis exécutables. Cela est possible grâce à des noyaux d'exécutif [75]. Ceux-ci contiennent la traduction en langage compilable de chaque macro. On différencie les noyaux génériques d'exécutif des noyaux applicatifs d'exécutif. Dans les premiers on trouve les macros réutilisables, c'est-à-dire celles correspondant aux services bas-niveau, ces noyaux pouvant servir pour une autre application sur le même type de processeur. Les deuxièmes contiennent les macros correspondant aux opérations de l'algorithme et dépendent donc de l'application. Il existe actuellement des noyaux génériques d'exécutif pour les machines Unix/Linux, les processeurs MPC555 de Motorola et i80386 d'Intel, les micro-contrôleurs i80C196 et MC68332, ainsi que quelques DSP (ADSP21060, TMS320C40, TMS320C60) et enfin pour les médias de communication TCP/IP et CAN. Parmi les services bas-niveau qu'ils permettent, l'un d'eux consiste à décrire le chargement de l'application. Il est ainsi possible de choisir un processeur maître à partir duquel les codes compilés seront envoyés aux autres processeurs.

Il existe également des travaux sur une version de SynDEx où les nœuds du graphe d'architecture ne sont pas des processeurs mais des circuits. La génération de code permet alors de générer du VHDL. Cette version du logiciel développée principalement à l'École Supérieure d'Ingénieurs en Électronique et Électrotechnique (ESIEE) s'appelle SynDEx-IC [76][77][78][79]. De plus, L'équipe AOSTE en collaboration avec l'équipe Pop-Art de l'INRIA Grenoble a réalisé une version du logiciel SynDEx permettant de faire de la tolérance aux pannes [80][81][82].

Le logiciel SynDEx a été ou est utilisé pour développer des applications réalistes par des industriels parmi lesquels on peut citer Thales, Nokia, PSA, Mitsubishi et MBDA. Par ailleurs de nombreuses applications ont été réalisées par des équipes de recherche académiques dans le domaine du traitement du signal et de l'image comme par exemple à l'INSA de Rennes sur des architectures multi-processeur à base de DSP [83][84]. Le logiciel SynDEx a également été employé pour des applications de contrôle-commande [85][86].

4.2 Le modèle flot de données conditionné dans le logiciel SynDEx

4.2.1 Description du conditionnement avec le langage SynDEx

La description graphique de l'algorithme avec le logiciel SynDEx utilise le langage du même nom. Ce langage reposant sur le modèle flot de données conditionné, il ressemble beaucoup à ce que nous avons présenté au chapitre 2. La figure 4.5 page 108 montre un algorithme conditionné

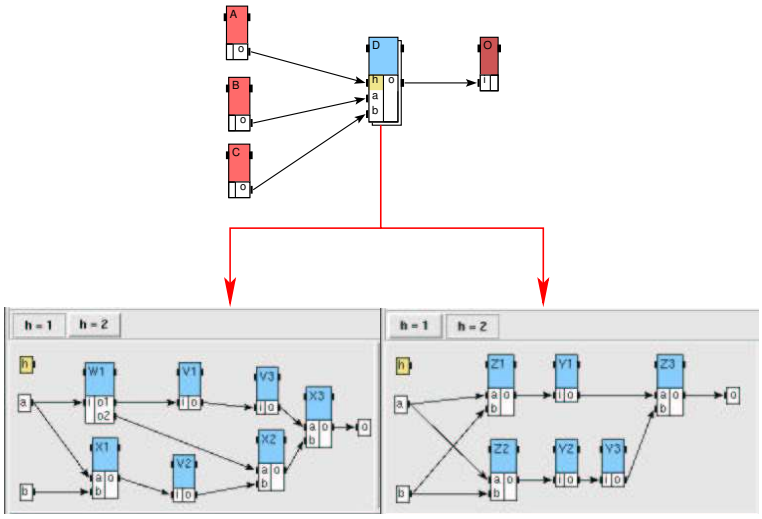

FIG. 4.5 – *Exemple d'algorithme conditionné décrit avec le logiciel SynDEx*

décrit avec le langage SynDEx en utilisant l'IHM du logiciel SynDEx. Le port de conditionnement est coloré en jaune (ici le port *h* de l'opération *D*). Les différents sous-graphes d'une opération conditionnante sont accessibles via un système d'onglets (visibles sur les deux graphes de la partie inférieure de la figure 4.5). Cet algorithme comporte donc une opération conditionnante *D* possédant deux sous-graphes d'opérations conditionnées.

4.2.2 Mise à plat du graphe SynDEx avant distribution

Ce n'est pas le graphe hiérarchique représentant l'algorithme qui est utilisé par les heuristiques d'ordonnancement et de distribution, mais le graphe obtenu par mise à plat de celui-ci. Cette mise à plat concerne le conditionnement (utilisation des méthodes décrites dans 2.2.3), mais aussi la hiérarchie simple (substitution d'un sommet par son sous-graphe) et les répétitions finies (substitution d'un sommet par *k* sommets identiques). La figure 4.6 page 112 montre le graphe obtenu après mise à plat du graphe d'algorithme de la figure 4.5 page 108. Dans le logiciel SynDEx, la mise à plat du graphe d'algorithme n'a lieu qu'à l'adéquation et le graphe obtenu n'est pas visible par l'utilisateur. Il existe néanmoins un mode `debug` permettant de générer un postscript représentant ce graphe. Chaque opération y est représentée par un rectangle divisé horizontalement en 3 parties : la partie supérieure comporte les ports d'entrée, la partie centrale contient le nom de l'opération et sa condition et la partie inférieure comporte les ports de sortie. Les arcs rouges sont les dépendances de conditionnement, les noirs sont les autres dépendances. Chaque arc est étiqueté par sa condition. Les étiquettes "$S : T$" et "$C0 : T$" signifient que leur condition est toujours vraie (T comme "True"). Sur la figure 4.6 page 112, les trois opérations capteurs sont situées à gauche (dans le sens de la page) et l'opération actionneur à droite. A droite des capteurs se trouvent les deux opérations *CondI* (ports *a* et *b* de *D*) et à gauche de l'actionneur se trouve l'opération *CondO* (port *o* de *D*). Entre ces opérations *CondI* et l'opération *CondO* se trouvent les opérations conditionnées corres-

pondant aux deux sous-graphes de l'opération conditionnante D. Celles correspondant au cas où $h = 2$ sont en haut et celles correspondant au cas $h = 1$ sont en bas. Sur la figure, h n'apparaît pas mais est remplacé par $A.o$ qui signifie "port de sortie o de l'opération A". En effet SynDEx nomme les dépendances de données en utilisant le nom du port producteur.

4.2.3 Distribution du conditionnement avec les heuristiques

Il convient ici de décrire comment la possibilité de spécifier du conditionnement modifie l'approche d'ordonnancement et de distribution énoncée dans [67]. La distribution consiste à choisir sur quel processeur une opération doit être exécutée et l'ordonnancement consiste à décider dans quel ordre sont exécutées des opérations distribuées sur un même processeur.

Certaines méthodes consistent à d'abord distribuer toutes les opérations et ensuite raisonner localement sur chaque processeur pour les ordonnancer. Dans SynDEx, l'ordonnancement et la distribution se font conjointement dans une approche constructive. A chaque pas de l'heuristique utilisée dans SynDEx, une opération est choisie parmi celles ordonnançables, puis est associée à un processeur (distribution) et y est ordonnancée. Cette heuristique est sans retour arrière (backtrack). L'ensemble des opérations ordonnançables est initialisé à l'ensemble des opérations sans prédécesseur. A chaque fois qu'une nouvelle opération a été distribuée et ordonnancée, elle est supprimée de l'ensemble des opérations ordonnançables. Les opérations dont, maintenant, tous les prédécesseurs ont été ordonnancés sont ajoutées à l'ensemble des opérations ordonnançables. L'ordonnancement et la distribution se poursuivent jusqu'à ce que l'ensemble des opérations ordonnançables soit vide.

La fonction de coût utilisée pour choisir l'opération à distribuer et à ordonnancer parmi l'ensemble des opérations ordonnançables utilise la "pression d'ordonnancement" définie dans [67]. Pour simplifier, il s'agit de l'allongement du temps d'exécution bout en bout estimé de l'algorithme que provoque une opération o distribuée et ordonnancée sur un processeur p. Plus la pression est grande, plus grand est l'allongement du temps d'exécution bout en bout estimé de l'algorithme et donc plus il est urgent d'ordonnancer l'opération. Cette pression prend en compte le coût des communications.

Afin de choisir la prochaine opération parmi l'ensemble des opérations ordonnançables, on utilise la pression d'ordonnancement pour associer à chacune des opérations son meilleur processeur (celui qui minimise sa pression) en tenant compte des communications à ajouter si les prédécesseurs de cette opération ont été distribués sur d'autres processeurs. Puis parmi les couples (opération, processeur) obtenus, on choisit le plus urgent, c'est-à-dire celui dont la pression est maximale. Cette méthode nécessite donc de connaître à chaque pas de l'heuristique les $M + P$ ordonnancements correspondant aux P processeurs et M médias de communication. Sur un processeur, les opérations sont ordonnancées dans l'ordre d'affectation par l'heuristique, la date de début au plus tôt d'une opération étant la date de fin de la dernière opération distribuée sur ce processeur.

L'introduction du conditionnement implique que deux opérations exclusives peuvent être ordonnancées sur le même processeur pendant le même intervalle de temps. En effet, lors de chaque répétition infinie du graphe motif, il n'y a exécution que de l'une des deux. Lorsque une opération est ordonnancée sur un processeur, sa date de début au plus tôt n'est plus la fin d'exécution de la dernière opération distribuée sur ce même processeur : il ne faut considérer que celles qui ne sont pas exclusives avec l'opération qu'on ordonnance. Pour un seul processeur, il faut donc non

plus tenir à jour un ordonnancement, mais k ordonnancement où k est le nombre de conditions [2] différentes rencontrées dans le graphe d'algorithme. Les communications pouvant elles aussi être exclusives, on passe donc de $M+P$ ordonnancements à $k(M+P)$ ordonnancements. L'impact sur la complexité des heuristiques est inférieur au facteur k, c'est-à-dire à la comparaison entre le nombre d'instructions nécessaires, avec un graphe d'algorithme sans conditionnement, pour une architecture de P processeurs et M médias de communication par rapport à une architecture de kP processeurs et kM médias de communication. En effet le nombre de pressions d'ordonnancement à calculer (correspondant à autant de distributions et ordonnancements "virtuelles" des opérations) n'augmente pas. Par contre, l'espace mémoire nécessaire à l'adéquation est bien multiplié par k à cause du nombre d'ordonnancements à tenir à jour.

La figure 4.8 page 113 montre le diagramme temporel résultant de l'adéquation de l'algorithme de la figure 4.5 avec l'architecture de la figure 4.7. Cette architecture est composée de deux processeurs de type MPC555 reliés par un bus CAN. Chaque colonne correspondant à un processeur ou un média de communication est divisée en sous-colonnes lorsque des opérations ou communications exclusives y sont ordonnancées durant le même intervalle de temps. C'est par exemple le cas sur le processeur $Proc1$ avec les opérations $Z2$ et $X1$ ou sur le processeur $Proc2$ où le groupe d'opérations $(W1, V1, V3)$ est exclusif avec le groupe d'opérations $(Z1, Y1)$. Lors de la génération de code, le logiciel SynDEx optimise alors le nombre de tests de conditionnement en profitant justement de ces regroupements d'opérations exclusives.

4.3 Avantages de SynDEx vis à vis des logiciels existants

Comme nous l'avons dit précédemment, la plupart des logiciels reposant sur des langages de programmation ont une vision monoprocesseur du problème. Ils permettent la description, la simulation et parfois la vérification de certaines propriétés du système. Mais les propriétés vérifiées, les performances constatées par simulation, ne sont valables que si l'implantation est monoprocesseur. Lorsque l'architecture est multi-processeur, il existe deux solutions. Soit les développeurs répartissent, avant de les décrire, les fonctionnalités sur l'architecture et peuvent ensuite considérer le système comme la somme de plusieurs systèmes monoprocesseurs et les simuler comme tels. Soit ils décrivent et simulent l'ensemble des fonctionnalités en monoprocesseur. Dans le premier cas, une implantation distribuée nécessite que les développeurs ordonnancent et distribuent eux-mêmes les communications entre les différentes fonctionnalités et donc les différents processeurs. Dans le second cas, au problème de l'ordonnancement et de la distribution des communications s'ajoute celui de la distribution et de l'ordonnancement des fonctionnalités. Dans les deux cas, ces décisions prises après simulation et vérification de propriétés peuvent conduire à un comportement du système différent de celui simulé ou même à des erreurs. SynDEx permet l'ordonnancement et la distribution automatique sur architecture distribuée d'un programme décrivant les fonctionnalités d'un système.

L'avantage principal de SynDEx sur la plupart des logiciels existant est donc de permettre l'ordonnancement et la distribution automatique sur architecture distribuée d'un programme décrivant

2. au sens défini dans 2.2.3.1, c'est à dire une suite ordonnée de conditions.

les fonctionnalités d'un système. De plus cette implantation prend en compte les architectures composés de processeurs de type différents ainsi que le routage et le coût des communications.

Il existe tout de même d'autres logiciels permettant de faire de la distribution de programmes, comme c'est le cas dans Ptolemy. C'est vis à vis de ces logiciels, que notre apport permet à SynDEx de proposer davantage. En effet, dans les logiciels permettant de faire de la distribution de programmes, les différentes alternatives d'un programme sont considérées comme atomiques lors de la distribution. Les programmes perdent alors une partie de leur parallélisme potentiel. En enrichissant le langage SynDEx avec le conditionnement ainsi que la mise à plat associée nous permettons au logiciel SynDEx d'exploiter complètement, si l'architecture le permet, le parallélisme potentiel des programmes y compris celui contenu dans leurs différentes alternatives.

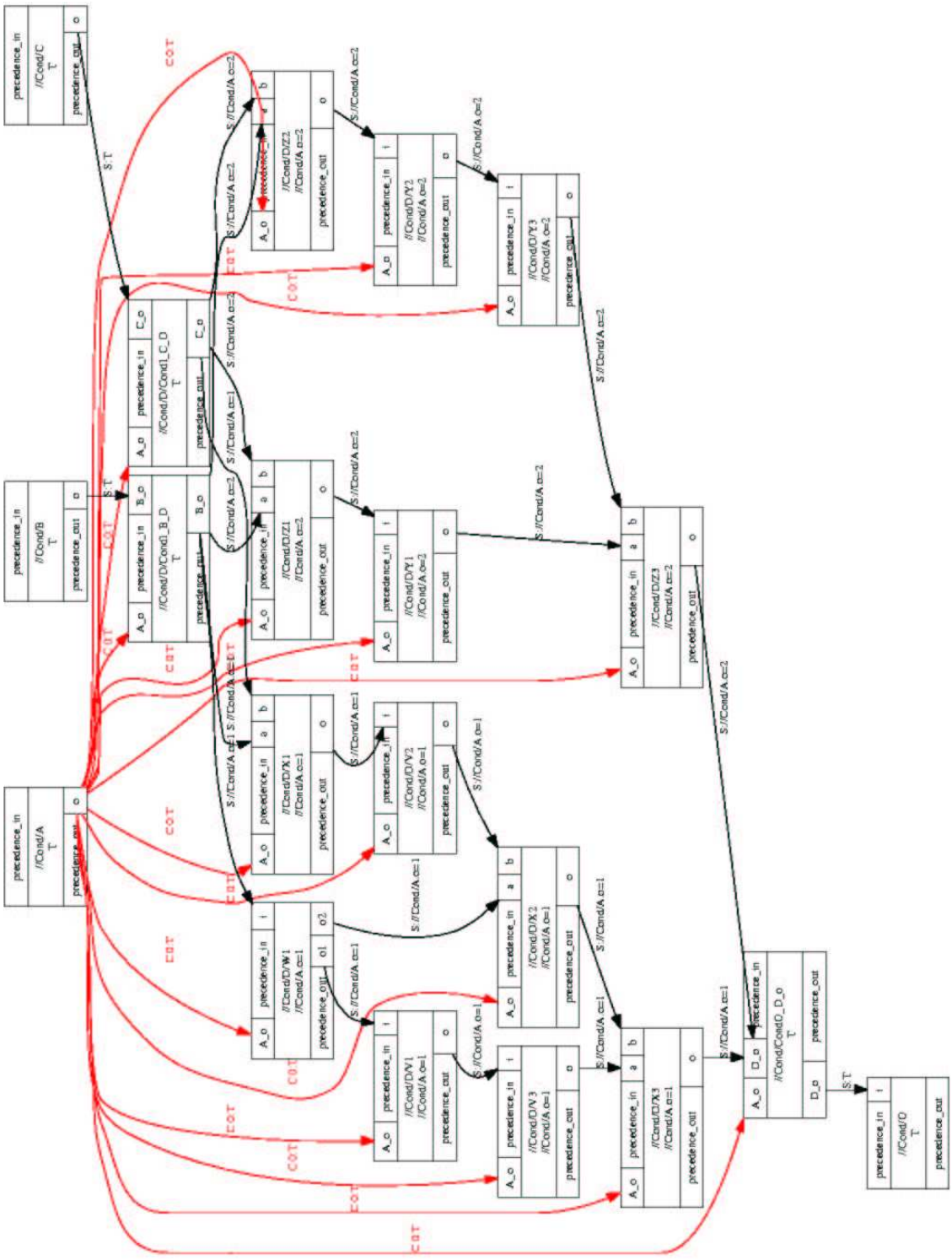

FIG. 4.6 – *Mise à plat par le logiciel SynDEx de l'algorithme de la figure 4.5 page 108*

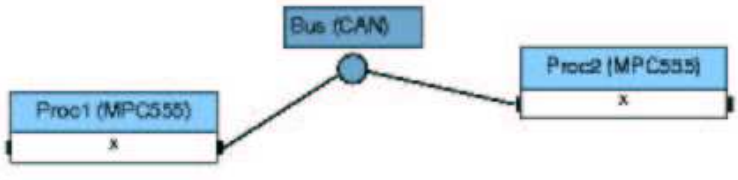

FIG. 4.7 – *Architecture décrite avec le logiciel SynDEx*

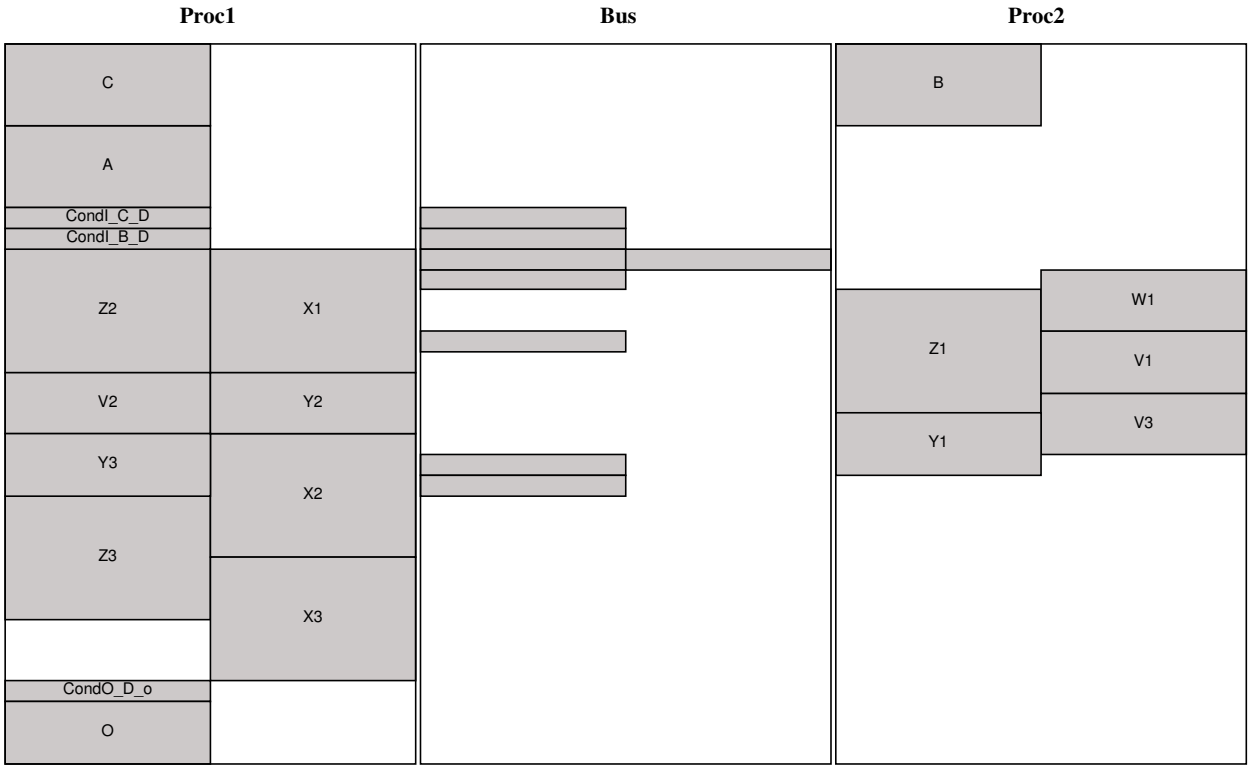

FIG. 4.8 – *Diagramme temporel résultant de l'adéquation de l'algorithme de la figure 4.5 avec l'architecture de la figure 4.7*

Chapitre 5

Exemples d'implantations distribuées de programmes conditionnés

Dans ce chapitre, nous donnons des exemples d'implantations distribuées de programmes conditionnés utilisant les principes énoncés dans les chapitres précédents. Nous consacrons d'abord une section à la traduction Scicos/SynDEx qui a été complètement implémentée dans la cadre du projet RNTL Eclipse. Nous décrirons également l'exemple industriel ayant servis à l'évaluation de cette traduction. Enfin nous donnerons un exemple de système où l'aglorithme est décrit à l'aide des langages SyncCharts et Scicos.

5.1 Traducteur Scicos/SynDEx

Dans le cadre du projet RNTL Eclipse réunissant PSA, CS, Crill Technology et l'INRIA, les principes de la traduction de programmes Scicos en programmes SynDEx énoncés dans la section 3.3 ont été implémentés dans une traduction complète [1] c'est à dire allant de l'utilisation des librairies existantes de blocs Scicos jusqu'à la génération de code distribué en utilisant SynDEx [2]. Ce travail a été réalisé par Cyril Faure, alors ingénieur expert dans le projet AOSTE de l'INRIA Rocquencourt. La figure 5.1 page 116 présente la chaîne de développement obtenue lors de l'utilisation de Scicos et du traducteur, puis de SynDEx. L'utilisateur peut ainsi :

- décrire son système (au sens automaticien du terme) en utilisant les bibliothèques de fonctions Scicos et le simuler,
- sélectionner une partie de son système et grâce au traducteur directement dans le compilateur de Scicos, générer un programme SynDEx,
- utiliser SynDEx pour décrire une architecture matérielle de simulation (hardware-in-the-loop) ou d'exécution embarquée,
- préciser des contraintes de distribution,

1. www.syndex.org/scicosSyndexGateway

2. En effet, le principes énoncés précédemment ne s'intéressaient pas aux fonctions associées aux blocs Scicos, hormis celles associées au blocs produisant des signaux d'activation.

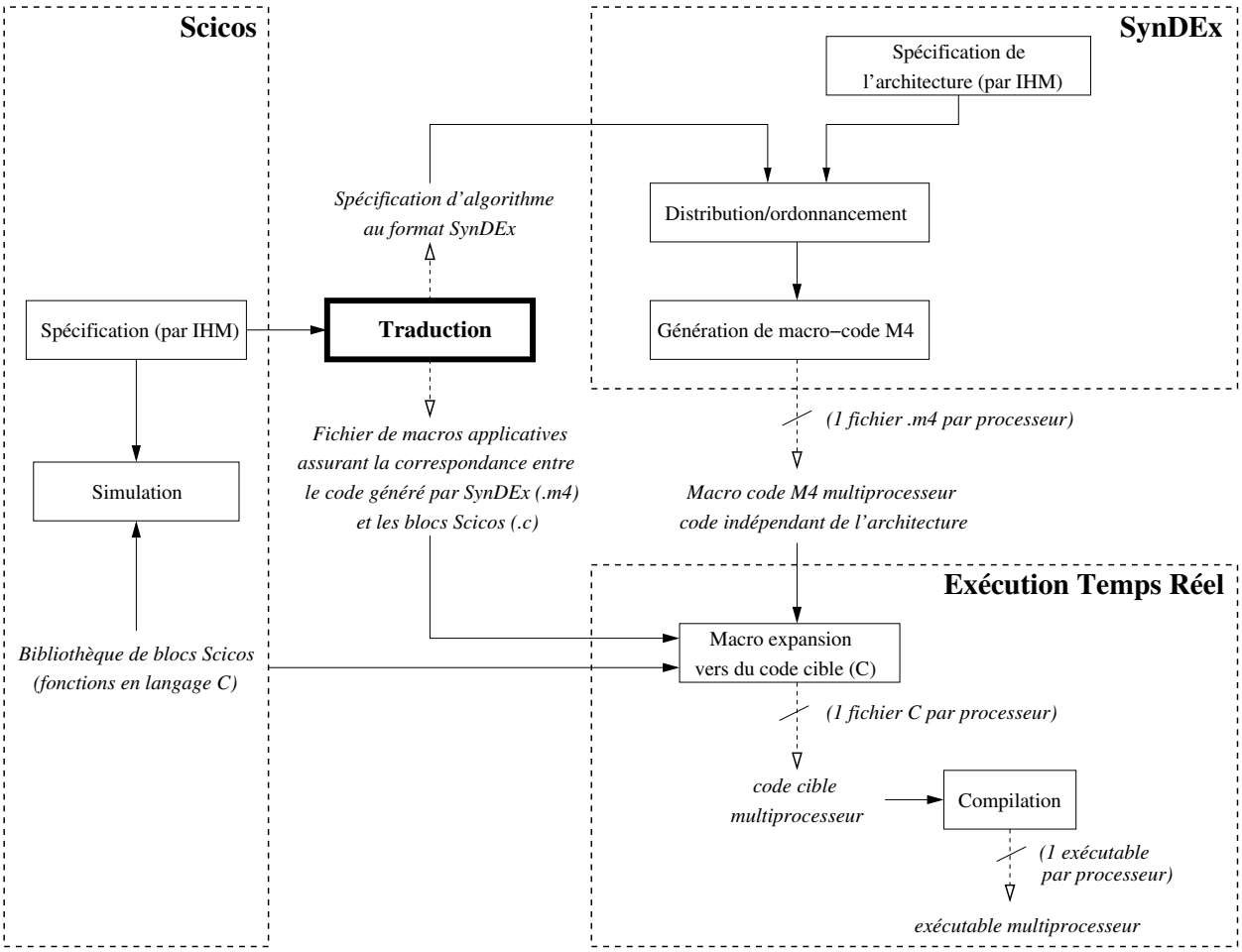

FIG. 5.1 – *Chaîne de développement de la modélisation/simulation jusqu'à l'implantation*

– utiliser les heuristiques d'adéquation de SynDEx pour obtenir une distribution et un ordonnancement temps réel satisfaisant,

– lancer la génération automatique de code de SynDEx, code que le compilateur lie automatiquement avec les librairies de fonctions de Scicos,

– compiler et exécuter le code en temps réel sur l'architecture spécifiée,

– exploiter les résultats obtenus et modifier éventuellement le programme Scicos initial.

Cette traduction permet de fournir une chaîne de développement cohérente allant de la modélisation du système jusqu'à son implantation distribuée. C'est cette génération de code distribuée qui apporte un plus par rapport aux outils classiques que sont Simulink associé à Real Time Workshop ou à dSPACE, ou bien Scicos et sa génération de code monoprocesseur.

5.2 Exemple d'implantation distribuée d'algorithme décrit avec Scicos

L'application industrielle utilisée pour évaluer le traducteur dans le cadre du projet Eclipse est une application automobile couplant anti-blocage des roues (ABS) et contrôle électronique de stabilité (ESP). Celle-ci a été développée par PSA en utilisant Scicos avec le traducteur et SynDEx. Le système est décrit avec Scicos par le programme illustré par la figure 5.2 page 117.

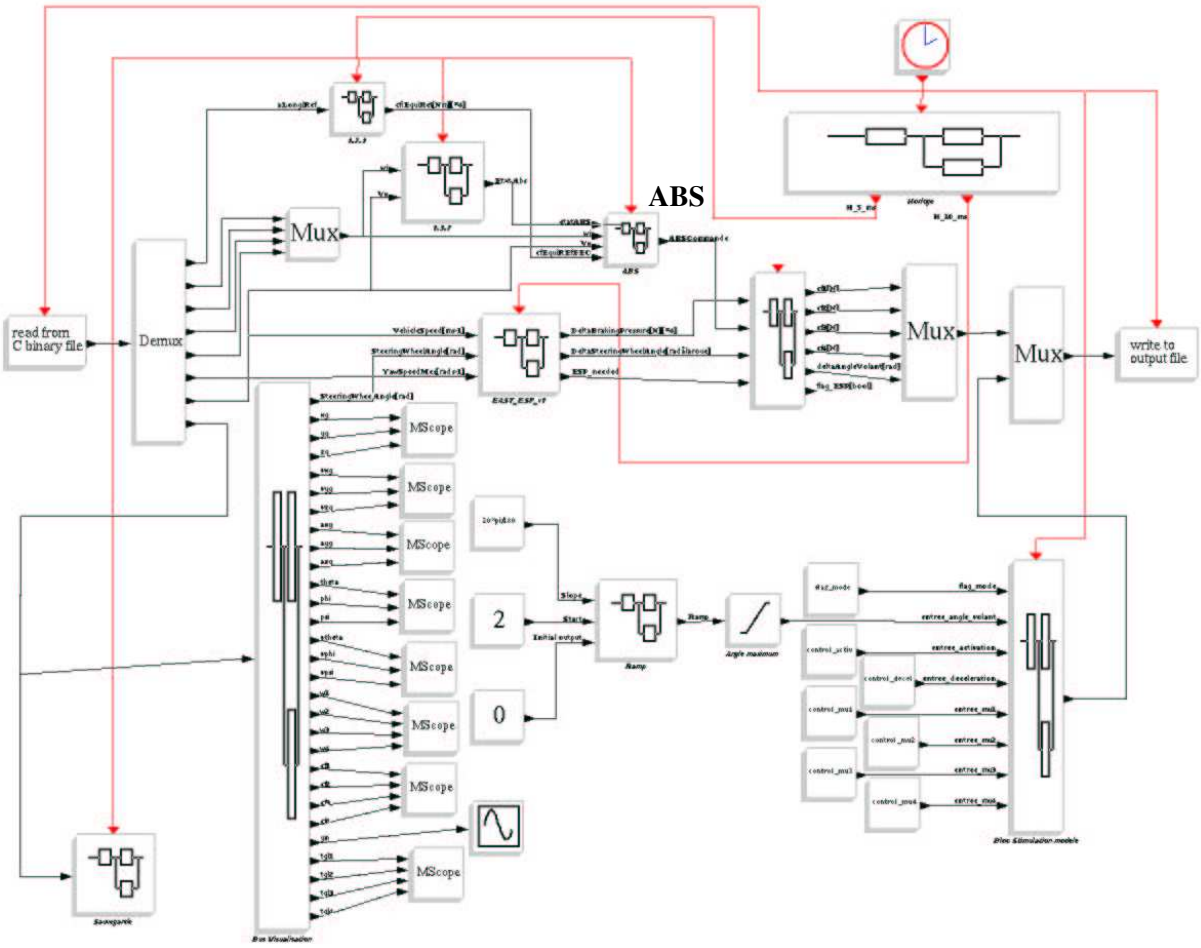

FIG. 5.2 – *Spécification Scicos d'une application automobile couplant ABS et ESP*

Le but est d'implanter le *superbloc* ABS sur une architecture distribuée. La description hiérarchique de ce bloc est donnée par la figure 5.3 page 121. Ce *superbloc* est sélectionné puis traduit directement par le compilateur Scicos. On obtient alors le programme SynDEx de la figure 5.4 page 122. On peut ensuite décrire un architecture et effectuer l'adéquation. La figure 5.5 page 122 montre l'adéquation de l'algorithme avec une architecture constituée de deux processeurs reliés par un bus CAN.

5.3 Exemple d'implantation distribuée d'algorithme décrit conjointement par SyncCharts et Scicos

5.3.1 Description du système

L'exemple considéré est un régulateur de vitesse pour automobile. Celui-ci, lorsqu'il est activé, permet au véhicule de garder une vitesse constante sans que le conducteur ait à maintenir une quelconque pression sur la pédale d'accélération. Ce système possède neuf entrées. Les six premières correspondant à des commandes manuelles placées sur le volant :

- *On* permet d'activer le régulateur de vitesse,
- Off permet de couper le régulateur de vitesse,
- *Set* permet d'initialiser le régulateur à la vitesse actuelle du véhicule. Celle-ci est ensuite maintenue par le régulateur tant qu'il est activé ;
- *Resume* permet de rétablir le régulateur de vitesse lorsqu'il a été momentanément désactivé par une pression de la pédale de frein ou de celle d'accélération. La dernière vitesse initialisée par la commande *Set* est alors maintenue à nouveau par le régulateur ;
- *QuickAccel* permet d'incrémenter la vitesse initialement enregistrée avec la commande *Set*,
- *QuickDecel* permet de décrémenter la vitesse initialement enregistrée avec la commande *Set*.

La pression sur les pédales d'accélération *Accel* et de frein *Brake* sont deux autres entrées. Enfin la vitesse actuelle du véhicule *Speed* constitue la dernière entrée.

Les sorties sont elles au nombre de cinq. Les affichages *RegOn*, $RegOff$ et *RegStby* correspondent aux différents états du régulateur : activé, désactivé et momentanément désactivé. L'affichage *CruiseSpeed* retourne la vitesse actuellement enregistrée par le régulateur (vitesse asservie). Enfin, le régulateur retourne une commande d'admission d'essence $ThrottleCmd$.

5.3.2 Description de l'algorithme avec SyncCharts et Scicos

La description du régulateur de vitesse utilise les langages SyncCharts et Scicos. La figure 5.6 page 123 montre le programme principal en langage Scicos. Celui-ci contient six blocs hiérarchiques, dont deux sont décrits en SyncCharts et quatre en Scicos. On a donc ici une utilisation conjointe de SyncCharts avec Scicos, comparable à l'utilisation de Stateflow avec Simulink.

Le bloc *PedalsPressed* permet de détecter si les pédales de frein ou d'accélération ont été pressées. Le programme Scicos correspondant est donné à gauche de la figure 5.7 page 123. De manière similaire, le bloc *SpeedLimit* compare la vitesse actuelle du véhicule aux vitesses minimales et maximales que le régulateur peut asservir. Le programme Scicos correspondant est donné à droite de la figure 5.7 page 123.

En fonction des signaux de sortie du bloc *SpeedLimit*, le bloc $SpeedFilter$ détermine si le régulateur peut asservir la vitesse actuelle du véhicule. Le programme SyncCharts correspondant est illustré par la figure 5.8 page 124.

L'état du régulateur (activation, désactivation, désactivation momentanée) est calculé par le bloc $CruiseState$ qui correspond au programme SyncCharts de la figure 5.9 page 124.

En fonction de l'état du régulateur et des commandes manuelles, le bloc *CruiseSpeedMgt* définit la vitesse à asservir *CruiseSpeed*. Le programme Scicos correspondant est donné par la figure 5.10 page 125. Enfin la régulation de vitesse est effectuée par le bloc *regulator* dont le programme Scicos est donné par la figure 5.11 page 125.

5.3.3 Traduction et unification des spécifications

Les programmes Scicos et SyncCharts sont ensuite traduits automatiquement en programmes SynDEx selon les règles énoncées dans les chapitres précédents. Les différents programmes SynDEx obtenus sont ensuite inclus dans des opérations SynDEx de niveaux hiérarchiques supérieurs qui sont ensuite connectées de manière semblable au programme Scicos de la figure 5.6 page 123. On obtient alors le programme SynDEx de la figure 5.12 page 126.

Les opérations *PedalsPressed* et *SpeedLimit* correspondent aux programmes Scicos homonymes. De même, les opérations conditionnantes *CruiseState* et *SpeedFilter* correspondent aux FSM SyncCharts homonymes. Chacune de ces opérations conditionnantes est connectée à autant d'opérations *delay* (vertes) qu'il y a de niveaux hiérarchiques dans la FSM correspondante (donc trois pour *CruiseState* et une pour *SpeedFilter*). Il est à noter que contrairement à l'usage de Stateflow dans Simulink où la FSM devient une fonction atomique, ici le programme SyncCharts devient un programme flot de données conditionné dont les opérations peuvent, elles aussi, être distribuées sur des processeurs différents. Les sorties de *CruiseState* servant de signaux d'activation aux programmes Scicos *CruiseSpeedMgt* et *Regulator*, les opérations SynDEx correspondantes sont des opérations conditionnantes. On retrouve également les capteurs et les actionneurs correspondant aux entrées et sorties du programme Scicos de la figure 5.6 page 123.

5.3.4 Implantation distribuée avec SynDEx

L'architecture distribuée sur laquelle doit s'exécuter cette application utilise trois processeurs : l'un se situe au niveau du pédalier, un autre au niveau du moteur et enfin un dernier au niveau du tableau de bord. Ces processeurs sont reliés par un bus CAN. L'architecture obtenue est donnée par la figure 5.13 page 126. L'acquisition de la pression sur les pédales de frein et d'accélération se font par des capteurs reliés au processeur du pédalier. La vitesse du véhicule est rendue disponible par le processeur du moteur, qui doit recevoir la commande moteur. Enfin, les commandes manuelles relatives au contrôleur ainsi que l'affichage de la vitesse asservie et de l'état du contrôleur, sont gérés par le processeur du tableau de bord. On utilise donc des contraintes de distribution pour "forcer" les heuristiques à distribuer les capteurs et les actionneurs du programme SynDEx sur les processeurs correspondants. La figure 5.14 page 127 montre une d'adéquation obtenue avec ces contraintes.

Les traductions de langages permettent ainsi de décrire les fonctionnalités en utilisant plusieurs langages. Si SynDEx et les traductions n'avaient pas été utilisés, la partie SyncCharts aurait été placée sur un seul processeur (les alternatives sont considérés comme atomiques) et il aurait fallu ordonnancer et distribuer les communications des actionneurs vers le processeur choisi, et de ce dernier vers les actionneurs. Or suivant l'état courant, toutes les acquisitions ne sont pas forcément nécessaires pour le calcul de la partie SyncCharts. De même, celle-ci ne produit pas forcément

dans toutes ses alternatives des données pour les actionneurs. Mettre en place ces communications de manière non automatique est ainsi difficile et hasardeux. Ici, la distribution et l'ordonnancement des communications ont été produits automatiquement par SynDEx. De plus, les communications générées tiennent compte des différentes alternatives décrites par le programme SyncCharts. Dans cet exemple, l'utilisation des traductions et de SynDEx a donc bien permis l'implantation distribuée de programmes conditionnés.

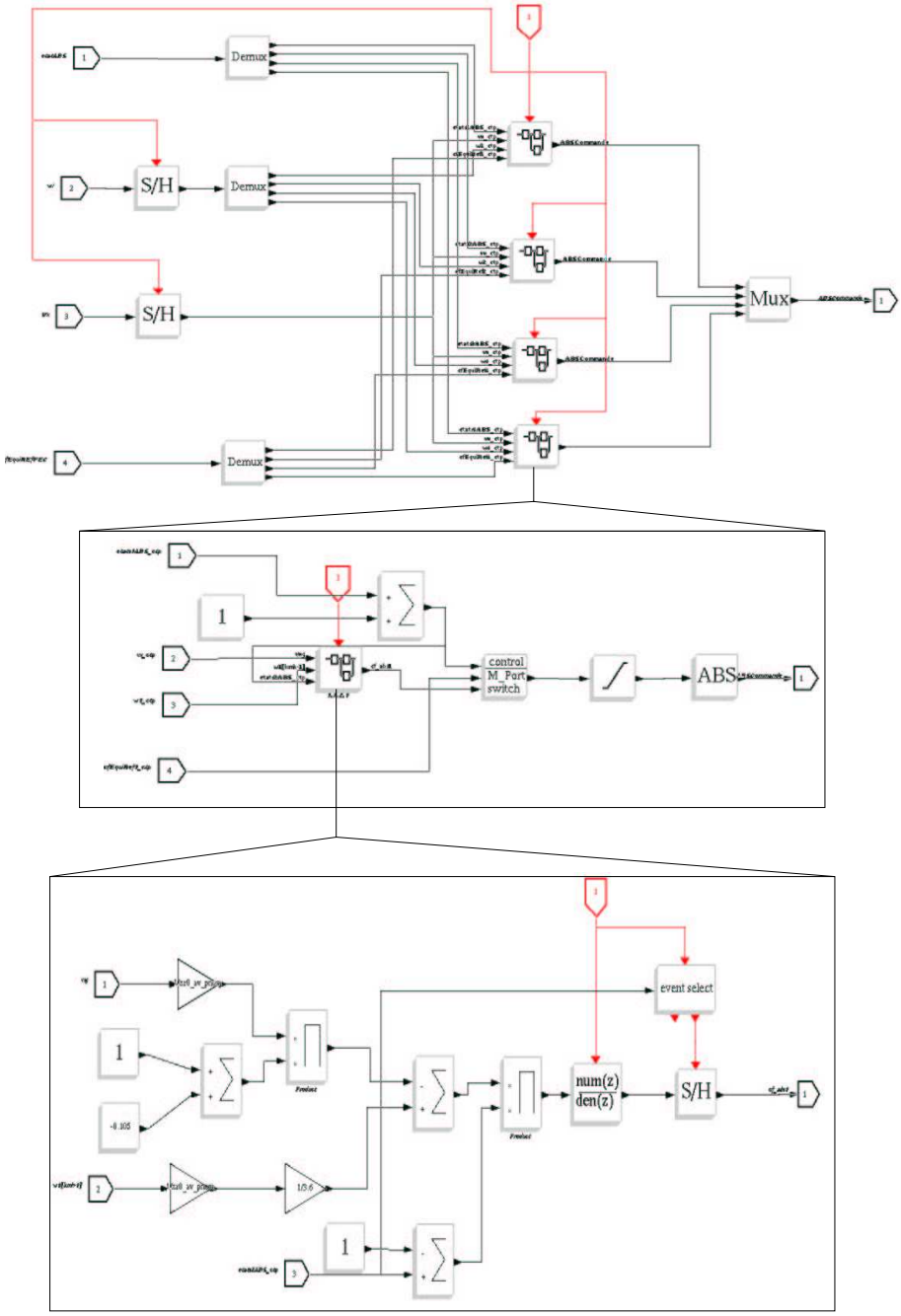

FIG. 5.3 – *Description hiérarchique du bloc ABS avec Scicos*

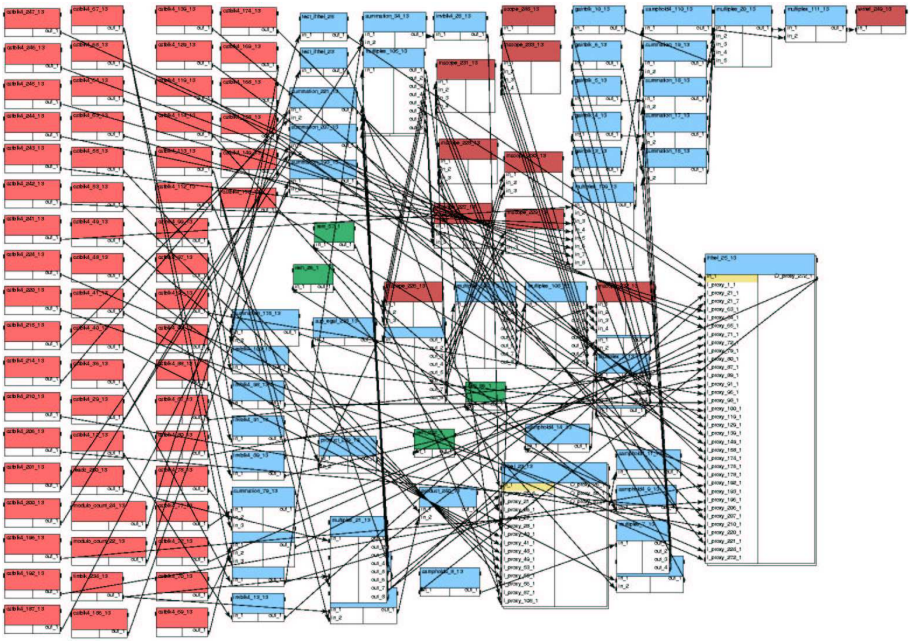

FIG. 5.4 – *Programme SynDEx obtenu après traduction du bloc Scicos ABS*

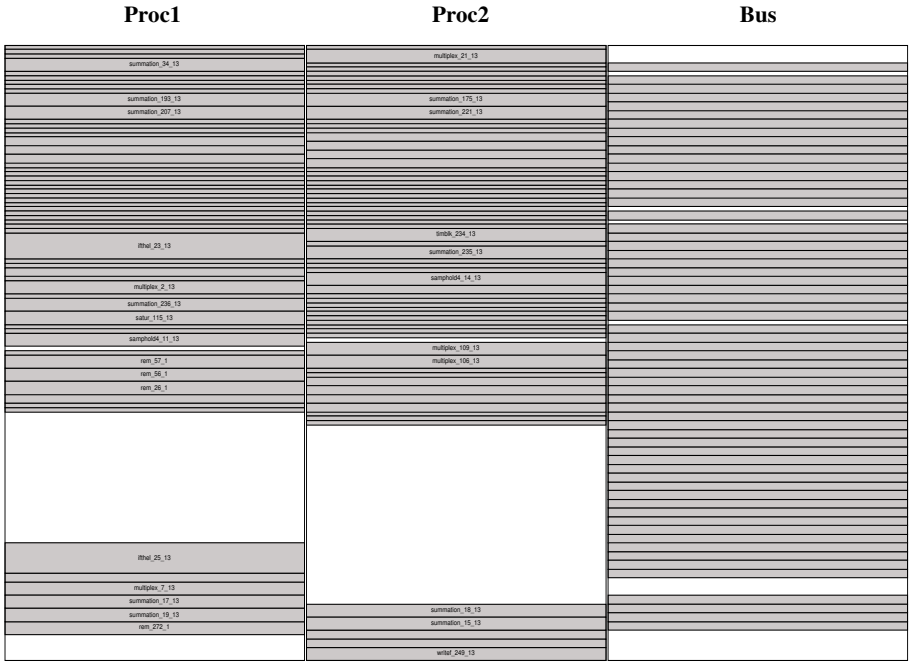

FIG. 5.5 – *Diagramme temporel résultant de l'adéquation de la traduction du bloc Scicos ABS avec une architecture constituée de deux processeurs reliés par un bus*

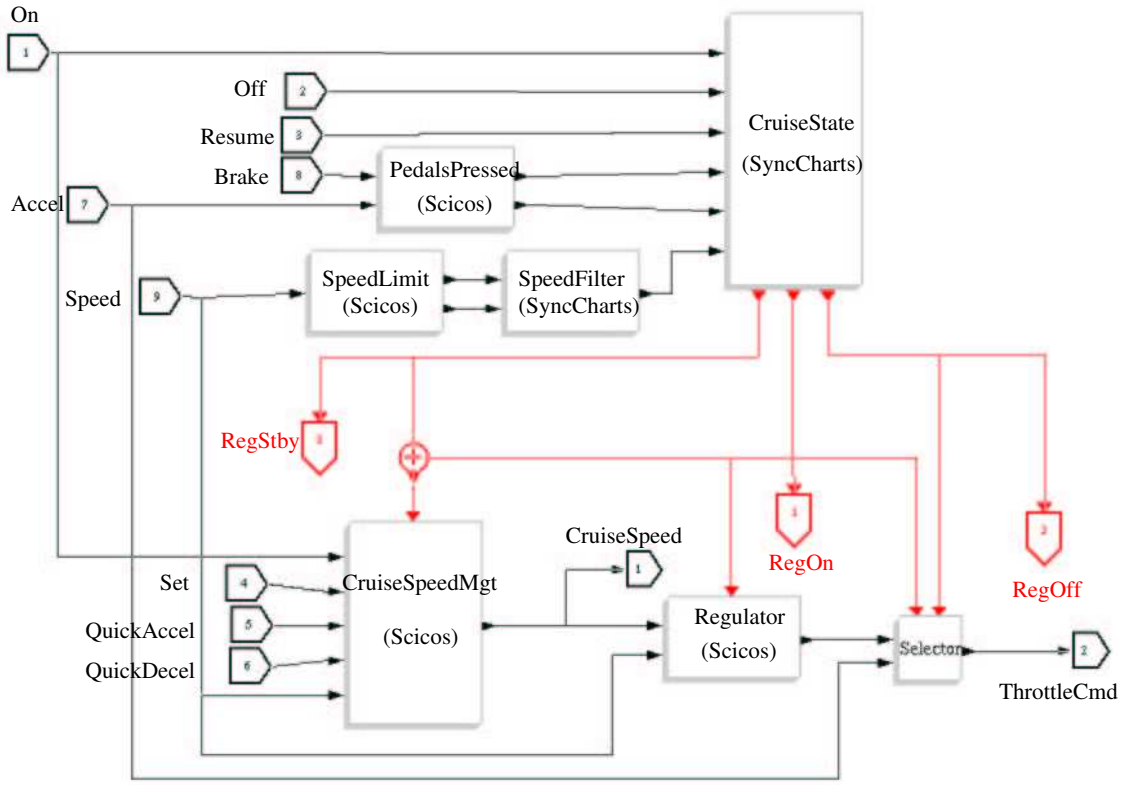

FIG. 5.6 – *Algorithme du régulateur de vitesse : programme principal (Scicos)*

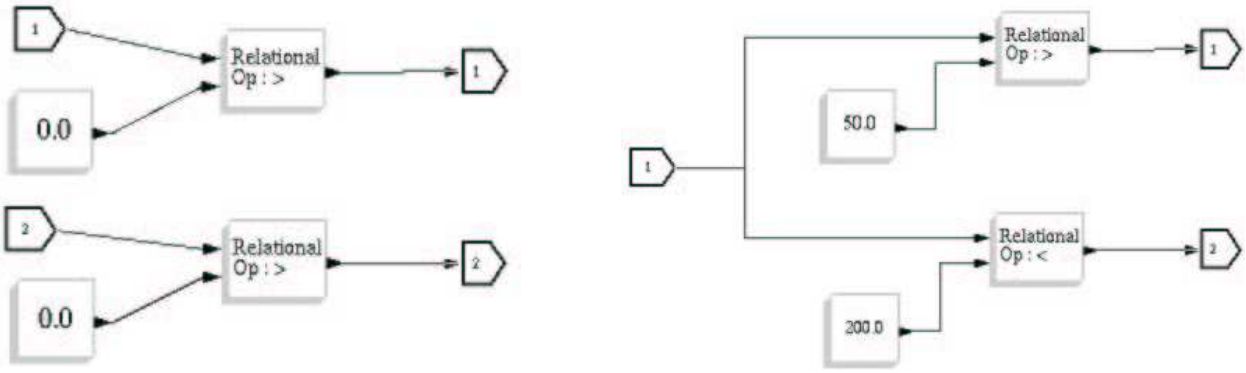

FIG. 5.7 – *Programmes Scicos correspondant aux blocs PedalsPressed (à gauche) et SpeedLimit (à droite)*

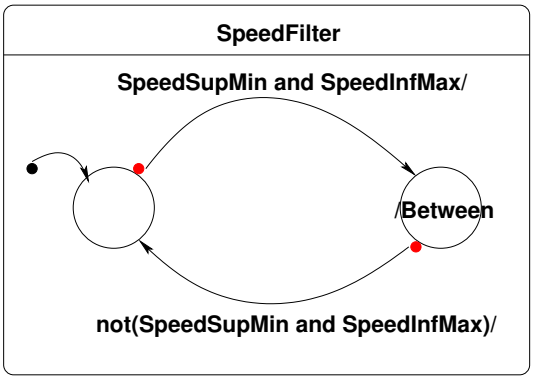

FIG. 5.8 – *Programme SyncCharts correspondant au bloc SpeedFilter*

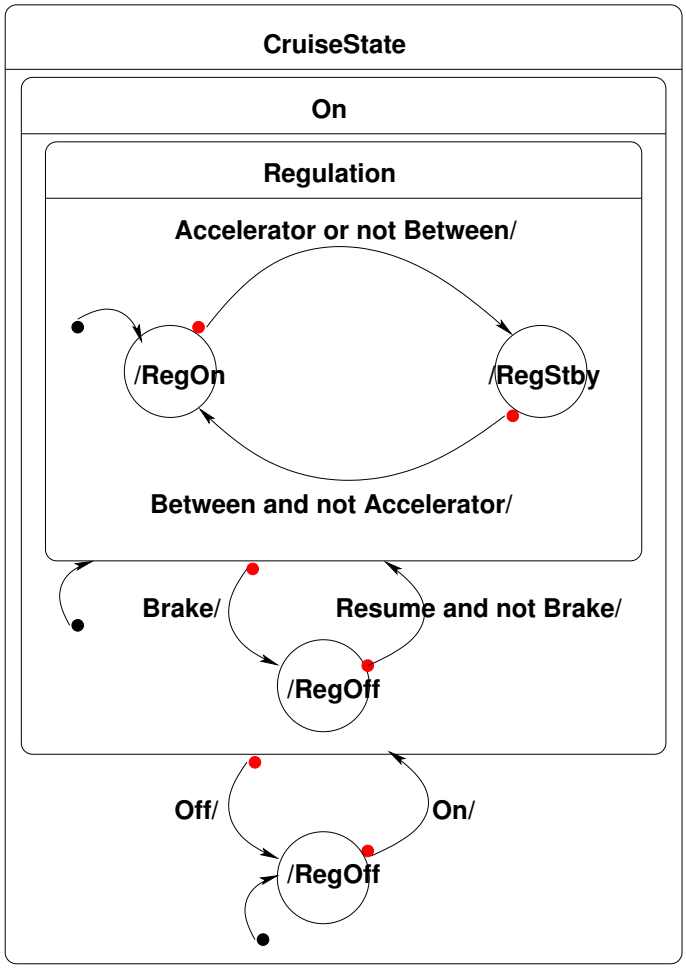

FIG. 5.9 – *Programme SyncCharts correspondant au bloc CruiseState*

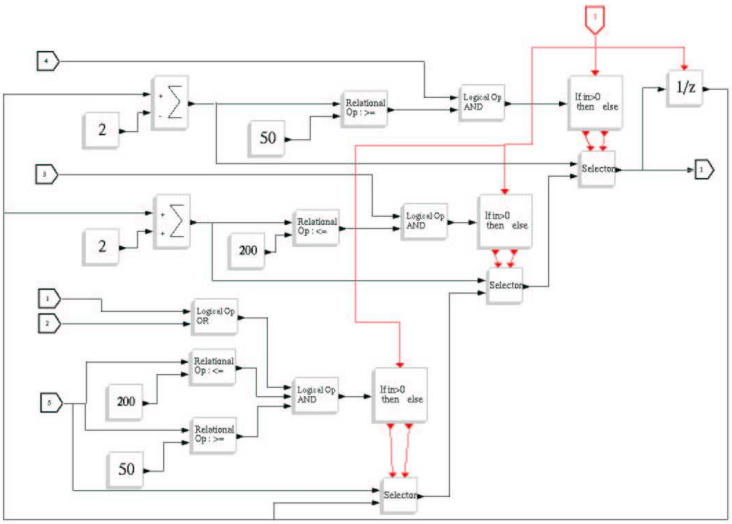

FIG. 5.10 – *Programme Scicos correspondant au bloc CruiseSpeedMgt*

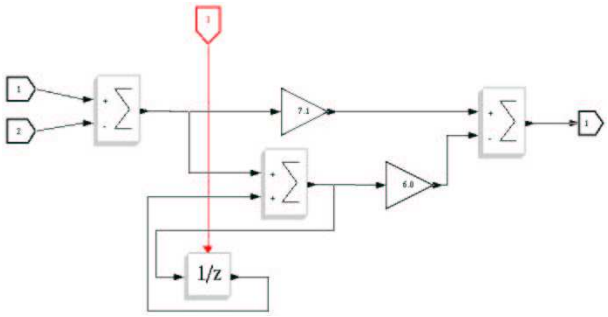

FIG. 5.11 – *Programme Scicos correspondant au bloc Regulator*

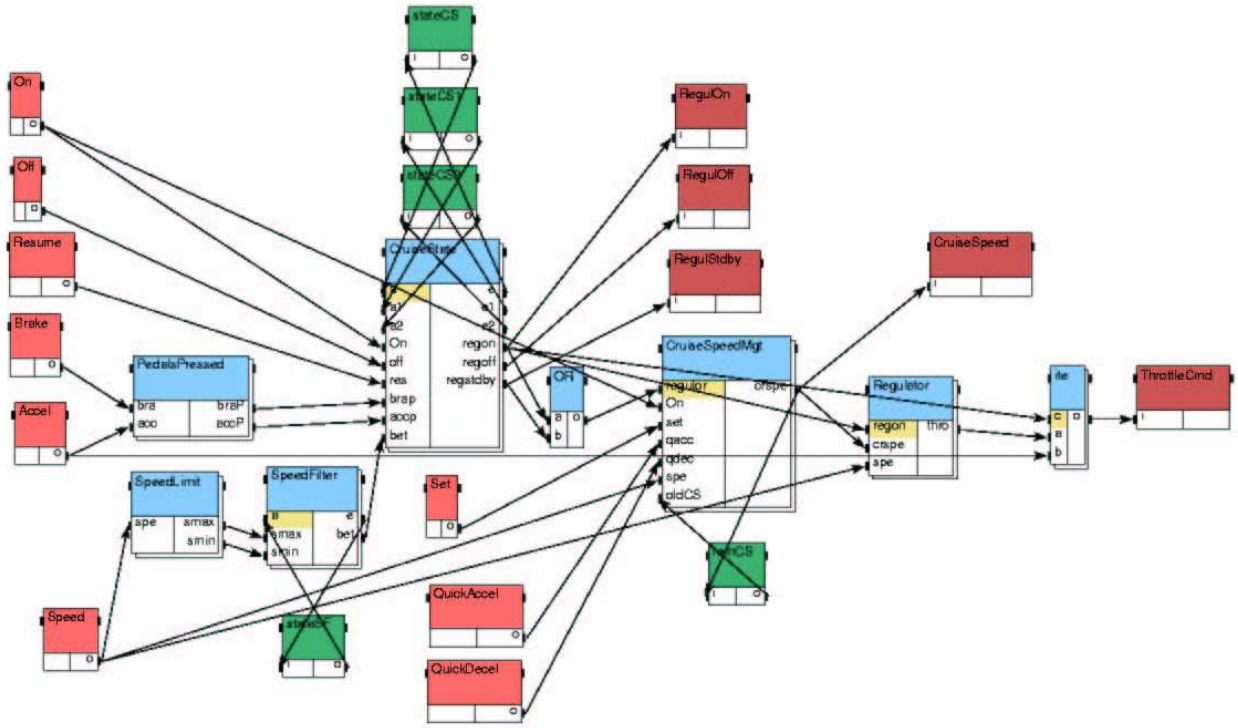

FIG. 5.12 – *Programme SynDEx obtenu après traduction des programmes Scicos et SyncCharts de l'algorithme du contrôleur de vitesse*

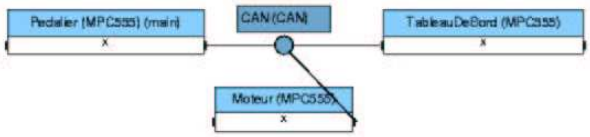

FIG. 5.13 – *Architecture spécifié avec SynDEx pour l'application de contrôleur de vitesse*

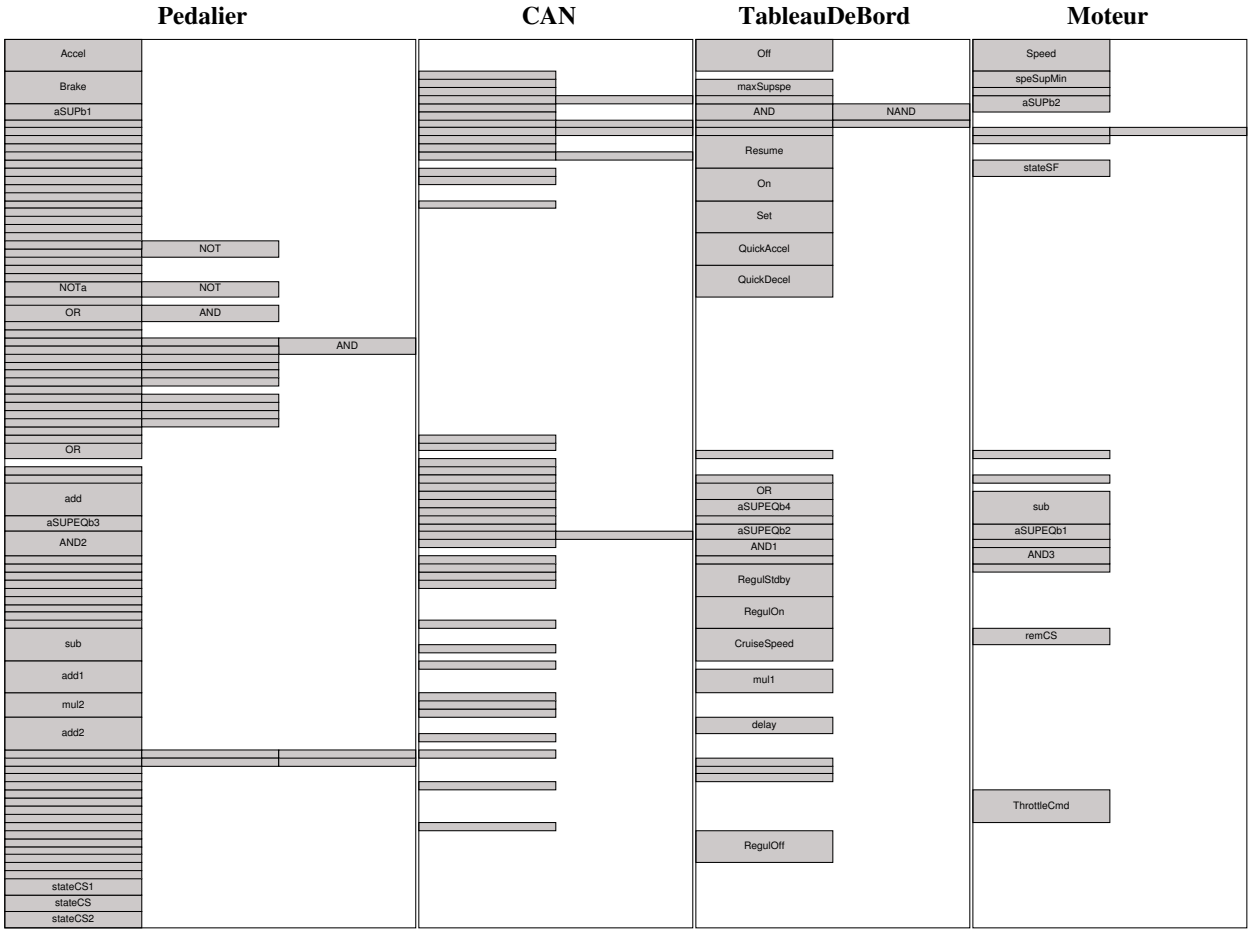

FIG. 5.14 – *Diagramme temporel résultant de l'adéquation de l'algorithme du contrôleur de vitesse de la figure 5.12 avec l'architecture de la figure 5.13*

Conclusion de la partie I

Le but de cette première partie était de proposer une méthode d'implantation distribuée de programmes conditionnés. Dans le cadre d'une implantation distribuée hors-ligne et déterministe, le conditionnement implique plusieurs cas d'exécution qu'il faut distribuer conjointement aussi bien sur les processeurs pour les opérations que sur les médias pour les communications résultant de l'implantation distribuée des opérations.

L'état de l'art a fait apparaître que les méthodes permettant l'implantation distribuée d'un ensemble de programmes sont rares, et dans celles existantes le conditionnement n'est pas pris en compte dans l'implantation distribuée. Ainsi un programme conditionné est soit directement considéré comme atomique, c'est-à-dire ne peut être découpé et distribué sur des processeurs différents (c'est le cas de Stateflow dans Simulink), soit transformé en un programme non conditionné d'un point de vue implantation distribuée (transformation de flot de données hétérogène en flot de données homogène comme dans Ptolemy). Nous avons observé que distribuer le conditionnement avec les langages flot de données, pourtant adaptés à l'implantation distribuée, n'est pas possible car la description du conditionnement ne respecte pas la règle d'activation du modèle flot de données. Dès lors certaines dépendances de données sont implicites et les communications conditionnées résultant du conditionnement ne peuvent être déduites du programme conditionné.

C'est pourquoi nous avons proposé un nouveau modèle flot de données conditionné inspiré des modèles flot de données hétérogènes existants qui est utilisé dans le langage SynDEx. L'intérêt de ce modèle est que les programmes l'utilisant peuvent être automatiquement modifiés de manière à rendre possible l'implantation distribuée du conditionnement sans passer par un langage flot de données homogène. Dans la représentation sous forme de graphe d'un tel programme, des sommets (opérations) et des arcs (dépendances de données) sont ainsi automatiquement rajoutés.

Si notre modèle s'avère adapté à l'implantation distribuée de programmes conditionnés, nous ne prétendons pas qu'il soit le mieux adapté à la description du conditionnement et à la simulation. En effet les développeurs de systèmes réactifs ont pour habitude d'utiliser principalement des langages FSM pour décrire le conditionnement ainsi que des langages utilisant le modèle schéma-bloc lorsqu'il s'agit de faire de la simulation. C'est en ce sens que nous avons proposé deux traductions de langages en langage SynDEx.

La première traduction permet de traduire des programmes SyncCharts en programmes SynDEx. SyncCharts est un langage de type FSM qui a l'avantage d'être déterministe. Néanmoins les principes de cette traduction sont utilisables pour d'autres langages de type FSM pour peu que la sémantique de ces langages puissent être réduite à une sémantique plus restreinte mais déterministe. C'est le cas de Stateflow, Statemate et Statecharts.

La deuxième traduction permet de traduire des programmes Scicos en programmes SynDEx. Scicos est un langage de type schéma-bloc qui se veut une alternative libre à Simulink. Le modèle schéma-bloc est proche du modèle flot de données mais ne respecte pas sa règle d'activation, et il n'y a pas réellement de dépendances de données puisque consommateur et producteur d'une donnée peuvent s'exécuter indépendamment. La traduction proposée peut être adaptée à d'autres langages de type schéma-bloc et a donné lieu à une implémentation complète, de manière à constituer une chaîne de développement où la traduction ainsi que l'implantation distribuée du programme en résultant sont entièrement automatisées.

Pour l'implantation distribuée de programmes conditionnés, on utilise le logiciel SynDEx dont le langage de description d'algorithme est le langage du même nom. Nous avons décrit comment SynDEx permet de distribuer le conditionnement. Enfin, nous avons donné deux exemples d'implantation distribuée de programmes conditionnés. Le premier exemple est une application industrielle réaliste que le groupe PSA a utilisée pour évaluer la traduction Scicos/SynDEx. L'autre exemple montre comment les deux traductions peuvent être associées ensemble. Une partie de l'application de régulateur de vitesse est ainsi décrite avec Scicos tandis que pour l'autre SyncCharts est utilisé. Une fois les différents programmes traduits, nous avons utilisé SynDEx pour obtenir automatiquement l'implantation distribuée des programmes obtenus. Cette utilisation conjointe de Scicos et SyncCharts peut être comparée à l'utilisation de Stateflow avec Simulink. Il existe néanmoins une différence importante. En effet, dans notre approche les programmes des deux langages sont traduits par des programmes flot de données conditionnés, alors que dans Simulink un programme Stateflow est traduit par une fonction atomique qui constitue ensuite un sommet d'un programme Simulink. Dans notre approche le parallélisme potentiel est sauvegardé alors que dans l'approche Simulink/Stateflow celui de la partie Stateflow est perdu.

Notre approche utilisant des traductions, notre modèle flot de données conditionné et SynDEx permet ainsi l'implantation distribuée de programmes conditionnés. Son apport est que le conditionnement est pris en compte dans la distribution et les dépendances de données conditionnées sont automatiquement générées.

Deuxième partie

Ordonnancement temps réel mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques

Introduction de la partie II

Cette deuxième partie est donc consacrée à l'ordonnancement mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques. On substitue ici le terme tâche au terme opération précédemment utilisé car on s'intéressera davantage aux caractéristiques temporelles qu'aux instructions constituant les programmes. Après avoir montré dans la première partie comment obtenir hors-ligne l'implantation distribuée de programmes conditionnés, on cherche à prendre en compte en-ligne des tâches apériodiques au comportement non déterministe. Pour cela, nous partons de l'hypothèse qu'un ordonnancement hors-ligne respectant des contraintes d'échéance et de latence a été calculé. Cet ordonnancement hors-ligne est supposé sans alternatives, c'est-à-dire ne comportant pas de tâches dont l'exécution est conditionnée. Si ce n'est pas le cas, on utilisera une version pessimiste de l'ordonnancement comportant des alternatives. Le but est ensuite de permettre au mieux l'exécution des tâches apériodiques sans remettre en cause le respect des contraintes temps réel des tâches périodiques.

Nous verrons que les nombreux résultats existants se restreignent à un modèle temps réel simple où la seule contrainte temps réel est l'échéance. Or, sans sur-contraindre le système, celle-ci ne permet pas de contraindre l'exécution d'une tâche relativement à la date d'exécution d'une autre. C'est pourquoi l'équipe AOSTE a introduit dans des travaux précédents la contrainte de latence qui permet d'imposer un délai entre la date de début d'exécution d'une tâche et la date de fin d'exécution d'une autre, ces tâches étant dépendantes. Nous montrerons que la contrainte de latence offre en-ligne davantage de flexibilité que l'échéance.

La complexité des calculs en-ligne nécessaires pour l'exécution de tâches apériodiques étant directement liée au nombres de contraintes à satisfaire, nous proposerons des règles de réduction permettant de garantir toutes les contraintes de l'ordonnancement hors-ligne en n'en considérant, en-ligne, qu'un sous-ensemble.

Ensuite nous choisirons d'étendre une des méthodes décrite dans l'état de l'art, appelée slot shifting, à l'ordonnancement mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques. Nous justifierons notre choix et proposerons deux solution différentes que nous discuterons en termes de complexité et d'optimalité.

Chapitre 6

État de l'art

6.1 Tâches périodiques, apériodiques et sporadiques

6.1.1 Définitions

Dans la suite par système, nous entendons système informatique et donc la combinaison architecture matérielle/programme. Ce programme se décompose en deux parties, l'application et le système d'exploitation servant de support à cette application.

La plupart des définitions données ici seront sûrement connues du lecteur. Néanmoins, on trouve, même pour ces termes de base du temps réel, une certaine divergence dans la littérature. Nous nous efforçons donc de rappeler chaque terme, afin que la suite du discours ne soit pas équivoque.

Une tâche* est une séquence d'instructions dont l'exécution nécessite l'allocation à une ressource ou processeur, c'est-à-dire son CPU mais aussi sa mémoire, ses entrées/sorties, etc. Une tâche, une fois allouée au processeur pour son exécution, occupe le processeur jusqu'à la terminaison de sa séquence d'instructions. Elle ne peut pas se suspendre elle-même, c'est-à-dire décider de stopper son exécution avant terminaison pour la reprendre plus tard. On ne parle de tâche que dans un environnement multi-tâche, c'est-à-dire où un ensemble de tâches sont en concurrence pour l'accès à une même ressource, le processeur. Afin de partager cette ressource, on est obligé de construire un ordonnancement, c'est-à-dire de définir un ordre d'exécution entre les tâches puisque deux tâches ne peuvent s'exécuter en même temps sur la même ressource.

Dans un système temps réel les tâches qui le composent sont dépendantes du temps. Par exemple l'exécution d'une tâche peut s'avérer inutile si elle a lieu après une certaine date. Ainsi, lorsque l'on considère des tâches temps réel, en plus de l'ordre des tâches dans l'ordonnancement on s'intéresse également aux dates d'exécution de celles-ci. On associe donc aux tâches temps réel des caractéristiques nécessaires pour construire l'ordonnancement. Les caractéristiques donnent des indications sur, par exemple, la durée d'exécution de la tâche et, si elle se répète, la durée séparant deux répétitions de la tâches, que cette durée soit fixe ou non. Par exemple les caractéristiques de la tâche T_a peuvent indiquer que cette tâche demande son exécution à la date $t = 2$ et nécessite 3 unités de temps pour son exécution. Tout ordonnancement où apparaît cette tâche doit donc prévoir son exécution après la date $t = 2$ et qu'elle soit allouée à un processeur durant 3 unités de temps.

En plus de ces caractéristiques, les tâches temps réel possèdent aussi des contraintes temps réel*. Les contraintes définissent parmi les ordonnancements possibles ceux qui sont corrects, c'est-à-dire qui respectent les contraintes. L'échéance* est un exemple de contrainte temporelle et indique la date au plus tard à laquelle une tâche doit avoir terminé son exécution. Si cette même tâche T_a possède une échéance à $t = 7$, on ne considérera comme corrects que les ordonnancements où l'exécution de T_a se termine au plus tard à cette date.

On parlera de temps réel strict* (*hard real-time*) quand le non-respect d'une de ces contraintes peut avoir des conséquences catastrophiques pour le système lui-même, par exemple sa destruction, ou pour son environnement, par exemple menacer des vies humaines. On parle au contraire de temps réel souple* ou mou (*soft real-time*) quand une contrainte temps réel non respectée dégrade le comportement du système sans remettre en cause son bon fonctionnement. s Puisque plusieurs tâches peuvent être en concurrence pour leur exécution par le processeur, il peut arriver qu'une ou plusieurs tâches soient en attente du processeur parce que celui-ci exécute une autre tâche. On appelle activation* d'une tâche l'instant où elle entre en concurrence avec d'autres tâches pour accéder à la ressource sans considération sur le fait que celle-ci puisse être occupée à exécuter une autre tâche. Cette activation peut être déclenchée par une interruption logicielle ou matérielle. Une tâche est active* jusqu'à ce qu'elle termine son exécution, puis devient inactive*. Dans la littérature, on parle parfois de tâche prête, ce qui est équivalent ici à active. Si une tâche se répète, il y a alternance d'intervalles de temps où la tache est active et d'intervalles de temps où elle est inactive.

Une tâche périodique* est une tâche qui est activée de manière répétée et pour laquelle deux activations successives sont séparées d'une durée constante appelée période* de la tâche. Habituellement, une tâche périodique possède une échéance, souvent définie relativement à la date de son activation.

Une tâche apériodique* est, étymologiquement, une tâche non périodique. Dans la littérature les définitions de ce terme diffèrent néanmoins. Le premier élément de divergence concerne le sens à donner à "non périodique". En effet, on peut considérer que c'est la propriété de répétition qui disparaît ou bien qu'il n'y a plus de durée constante entre les activations répétées. Dans certains articles, on précise clairement, comme dans [87], qu'une tâche apériodique n'est activée qu'une seule fois. Dans d'autres articles cela n'est pas indiqué et on peut penser qu'une telle tâche peut être activée une infinité de fois, éventuellement en un temps borné. On peut donc avoir deux points de vu qui sont équivalents. Ou bien on considère qu'un ensemble de tâches apériodiques est un ensemble de N tâches se répétant chacune éventuellement un nombre infini de fois, ou bien que c'est un ensemble de M tâches distinctes ne se répétant pas, M pouvant être infini. N'ayant aucune information sur la durée séparant d'éventuelles répétitions, le fait même qu'une tâche apériodique puisse se répéter ne permet pas de prendre davantage de décisions, pour son ordonnancement, que dans le cas où l'on considère qu'elle ne se répète pas. Avec ce type de tâche, on ne peut prendre une décision (ordre et date d'exécution) qu'à sa date d'activation, sans considération sur ce qui va arriver ensuite, l'information de répétition étant donc, pour son ordonnancement, totalement inutile. D'autre part, il y a des applications (client-serveur par exemple) où les tâches ne peuvent être caractérisées qu'à leur activation, chaque tâche étant différente de la précédente. Nous choisirons donc dans ce manuscrit de définir les tâches apériodiques comme des tâches ne se répétant pas.

Néanmoins, afin de mener un raisonnement objectif sur les autres éléments de divergence, nous considérerons le problème ouvert jusqu'à la fin de cette section.

Le deuxième élément de divergence concerne les dates d'activation des tâches apériodiques. Si la problématique des tâches apériodiques se concentre actuellement sur des tâches dont on ne connaît pas la date d'activation, cette particularité est rarement précisée, comme si cela était naturel pour de telles tâches. Or, dans son ouvrage [5], Buttazzo référence sur la problématique des tâches apériodiques des travaux anciens, voire antérieurs à la définition des termes "tâche périodique" et "tâche apériodique", dans lesquels les auteurs traitent indifféremment du cas où les dates d'activation sont connues [88] ou non [89].

Enfin, le troisième élément de divergence concerne la signification d'une échéance pour une tâche apériodique. Dans le cas où il n'y a pas d'échéance, nous parlerons de tâche apériodique souple* (*soft aperiodic task*) [90]. Mais dans le cas contraire, l'existence d'une échéance peut signifier différentes choses, selon l'interprétation des deux premiers éléments de divergence. Si on connaît les dates d'activation des tâches apériodiques celles-ci peuvent avoir des échéances, le problème de savoir si on peut garantir l'exécution des tâches avant leurs échéances reste soluble et garde un intérêt. Si les dates d'activation sont inconnues mais que l'ensemble des tâches apériodiques est connu avant exécution et que ces tâches ne sont activées qu'une seule fois, le problème reste soluble. Par contre si on ne connaît pas l'ensemble des tâches apériodiques ou que celles-ci peuvent se répéter (ce qui revient en fait au même) on ne peut plus rien garantir. En effet, soit une tâche apériodique se répétant, avec une durée d'exécution C et une échéance D signifiant que la tâche doit avoir fini son exécution au plus tard D unités de temps après son activation. Il suffit que les activations successives soient séparées d'une durée k inférieure à la durée d'exécution pour qu'immanquablement la n^{ième} itération de la tâche, avec $n \in \mathbb{N}$ tel que $(n-1)(C-k) \leq D < n(C-k)$, ne satisfasse pas son échéance.

En conséquence, décider si on peut garantir l'exécution d'un ensemble de tâches apériodiques se répétant et aux dates d'activation inconnues avant leurs échéances n'est possible que si on a connaissance, pour chaque tâche apériodique d'une durée minimum d'inter-activations* (*minimum interarrival time*), c'est-à-dire séparant deux activations successives. Ces tâches sont alors appelées tâches sporadiques* (*sporadic task*) [91][92]. Parfois, une tâche sporadique est définie comme étant une tâche apériodique qui à une échéance temps réel strict [93][94]. Il faut alors comprendre implicitement que l'échéance fait également office de durée minimum d'inter-activations.

Dans le cas où il n'y a aucune information ni sur cette durée minimum d'inter-activations ni sur les dates d'activation, il n'est pas possible de garantir l'exécution d'une tâche apériodique avant son échéance. On peut, par contre, se poser la question de savoir lorsque la tâche est activée, s'il est possible de l'exécuter avant son échéance. Si c'est le cas on accepte cette tâche et on garantit le respect de son échéance. Sinon, cette activation de la tâche apériodique est rejetée. Ce type de tâche sera appelé tâche apériodique ferme* (*firm aperiodic task*) [90]. Afin d'accepter ou de rejeter une telle tâche, on effectue un test d'acceptation*.

Nous rappelons que les définitions que nous donnons sont une synthèse de celles trouvées dans la littérature. Par la suite nous utiliserons ces définitions, même lorsque nous ferons référence à des articles où d'autres sont utilisées. En cas de difficulté, le lecteur pourra se reporter à l'index pour

savoir à quelle page chaque terme est défini. Pour résumer, les quatre types de tâches sont :

- tâche périodique : tâche se répétant et dont les activations successives sont séparées d'une durée constante appelée période ;
- tâche sporadique : tâche pouvant se répéter mais dont la durée séparant deux activations successives ne peux pas être inférieure à sa durée minimum d'inter-activations ;
- tâche apériodique ferme : tâche activée au plus une seule fois possédant une échéance. Cette échéance doit être respectée ou la tâche non exécutée ;
- tâche apériodique souple : tâche activée au plus une seule fois ne possédant pas d'échéance.

Les contraintes des deux premiers types peuvent être temps réel strict ou souple, par contre les deux types de contraintes suivants sont forcément temps réel souple.

6.1.2 Modèle temps réel classique et notations

Nous allons ici présenter le modèle temps réel classique et ses notations. Ces dernières variant d'un article à un autre, nous avons choisi de définir dès à présent les notations qui seront utilisées dans le manuscrit. Elles sont largement inspirées de travaux précédents ayant eu un soucis de concensus, comme l'ouvrage de G. Buttazzo [5] ou l'article collectif [95]. Nous avons indiqué à côté de chaque terme sa traduction en anglais. Par convention, pour les notations faisant référence au temps, les lettres majuscules désignent des durées alors que les minuscules désignent des dates. De plus, toutes ces dates et durées sont des entiers. Cette approche s'explique par le fait que ce temps sera, après implémentation, mesuré par un processeur ayant une fréquence d'horloge fixe, la période correspondante définissant une unité de temps indivisible.

6.1.2.1 Tâche périodique

Une tâche périodique est activée une infinité de fois. A chaque activation correspond une répétition de la tâche appelée instance*. Voici les notations relatives à une tâche périodique T_i pour $i \in \mathbb{N}$:

- la $j^{ème}$ instance de la tâche T_i se note $\tau_{i,j}$;
- on appelle C_i la durée d'exécution au pire cas* (WCET pour *Worst Case Computation Time*) de la tâche T_i ;
- P_i désigne la période* de la tâche T_i ;
- $r_{i,j}$ est la date d'activation* (*release time*) de $\tau_{i,j}$. Les dates d'activation de deux instances successives d'une tâche T_i sont séparées d'exactement P_i unités de temps. Ainsi $r_{i,j+1} = r_{i,j} + P_i$;
- on appelle Φ_i la phase* de la tâche T_i. C'est la durée séparant la date 0 de la première date d'activation de T_i. On a $\Phi_i = r_{i,1}$ et donc $r_{i,j} = \Phi_i + (j-1)P_i$;
- on note D_i la contrainte d'échéance (*deadline*). Elle définit l'échéance relative*, c'est-à-dire la durée maximum pouvant séparer la date d'activation d'une instance de la tâche T_i et la fin de son exécution. Cette échéance relative définit pour chaque instance de la tâche T_i une échéance absolue*. Cette échéance absolue est la date à laquelle l'exécution de cette

instance doit être finie. L'échéance absolue de $\tau_{i,j}$ se définit ainsi : $d_{i,j} = r_{i,j} + D_i = \Phi_i + (j-1)P_i + D_i$;

– $s_{i,j}$ désigne la date de début d'exécution* (*start time*) de $\tau_{i,j}$. De même $f_{i,j}$ désigne la date de fin d'exécution* (*finish time*) de $\tau_{i,j}$.

Ainsi, une tâche périodique T_i est caractérisée par un quadruplet (Φ_i, C_i, D_i, P_i). Ces quatre durées suffisent à caractériser l'infinité d'instances de la tâche T_i. On appellera jeu de tâches* un ensemble de tâches aux caractéristiques connues.

La figure 6.1 page 139 illustre les notations dans le cas d'une tâche périodique.

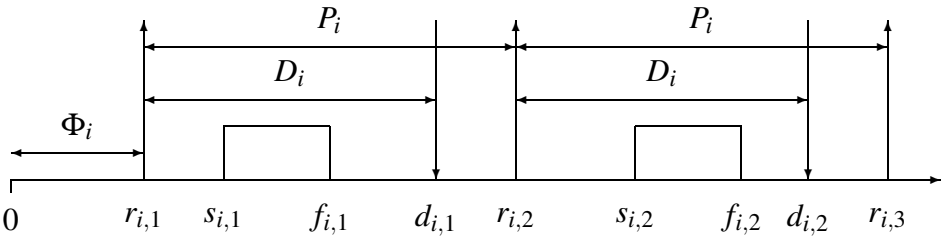

FIG. 6.1 – *Représentation graphique du modèle temps réel classique*

6.1.2.2 Tâche apériodique

Une tâche apériodique J_i, $i \in \mathbb{N}$ est caractérisée par son unique date d'activation r_i qui est souvent une inconnue du problème. Dans le cas d'une tâche apériodique ferme* (*firm*), la tâche survenant à cette date possède deux autres caractéristiques, à savoir sa durée d'exécution au pire cas C_i et une échéance absolue d_i. Elle est donc notée $J_i(r_i, C_i, d_i)$. S'il s'agit d'une tâche apériodique souple* (*soft*), C_i est optionnel, et il n'y a pas d'échéance.

6.1.2.3 Contraintes de précédence

En plus de ces caractéristiques, les tâches peuvent avoir des contraintes de précédence. Une contrainte de précédence de la tâche T_i sur la tâche T_j, notée $T_i \rightarrow T_j$ implique que la tâche T_i doit avoir terminé son exécution avant que T_j ne débute la sienne. Si les tâches T_i et T_j sont périodiques, cela implique $f_{i,k} \leq s_{j,k} \quad \forall k \in \mathbb{N}$. De telles contraintes sont utilisées pour spécifier, par exemple, une dépendance de données entre deux tâches.

La spécification de telles contraintes mène à représenter l'ensemble des tâches sous la forme d'un graphe de précédence*. Ce graphe est un graphe orienté acyclique (*DAG*) où chaque nœud est une tâche et chaque arc est une contrainte de précédence de la tâche source sur la tâche destination. Il est à noté qu'un graphe de précédence peut contenir des composantes non connexes, c'est-à-dire plusieurs sous-graphe ne possèdant pas de contraintes de précédence entre eux. L'ensemble des arcs d'un graphe de précédence définissent un ordre partiel sur l'ordre d'exécution des tâches, comparable à celui impliqué par les arcs d'un graphe flot de données.

Soit un graphe de précédence de n tâches, $n \in \mathbb{N}$. On appelle successeurs directs* d'une tâche T_i de ce graphe l'ensemble des tâches T_k, $k \in [1,\ldots,n]$, telles que $\exists\, T_i \rightarrow T_k$. On dit que T_j est un successeur* de T_i s'il existe une suite de tâches $T_i,T_x,\ldots,T_y,T_j$ telles que chaque tâche de la liste est un successeur direct de celle qui la précède. De même on nomme prédecesseurs directs* d'une tâche T_i l'ensemble des tâches T_k telles que $\exists\, T_k \rightarrow T_i$. On dit que T_j est un prédecesseur* de T_i s'il existe une suite de tâches $T_j,T_x,\ldots,T_y,T_i$ telles que chaque tâche de la liste est un prédecesseur direct de celle qui le suit. Une tâche sans prédecesseur est dite tâche d'entrée*, une tâche sans succésseur est dite tâche de sortie*.

6.1.2.4 Ordonnancement et ordonnançabilité

On appelle ordonnancement* une séquence d'instances de tâches pour lesquelles on a défini un ordre et des dates d'exécution, c'est-à-dire les intervalles de temps durant lesquels elles occuperont le processeur.

Un ordonnancement est dit correct* ou valide si l'ordre et les dates d'exécution des instances de tâches permettent de satisfaire toutes les contraintes temps réel du jeu de tâches.

On dit qu'un jeu de tâches est ordonnançable* s'il existe un ordonnancement correct de ce jeu de tâches.

Statuer sur l'ordonnançabilité* d'un jeu de tâches consiste à savoir si ce jeu de tâches est ordonnançable ou pas.

6.1.3 Ordonnancement de tâches périodiques

Étant donné un jeu de tâches périodiques, le problème consiste à décider de son ordonnançabilité sur un processeur. Ce type d'ordonnancement est dit monoprocesseur par opposition à l'ordonnancement multi-processeur ou distribué. Dans la suite, les résultats présentés ne concernent, sauf si cela est précisé explicitement, que le cas monoprocesseur.

Il existe différents algorithmes d'ordonnancement permettant d'ordonnancer des tâches périodiques. On dit qu'un algorithme d'ordonnancement de tâches périodiques est optimal* dans une classe d'algorithme si, quand cet algorithme ne trouve pas d'ordonnancement correct, aucun algorithme de la même classe ne permet d'en trouver un.

On différencie d'abord les ordonnancements préemptifs* des ordonnancements non préemptifs*. Dans un ordonnancement préemptif, une tâche ayant débuté son exécution peut être interrompue afin que le processeur exécute une autre tâche. Si on revient au modèle de tâches présenté par la figure 6.1 page 139, une tâche ne s'exécute pas forcément d'un seul bloc. Son exécution peut être morcelée comme le montre la figure 6.2 page 141. Dans ce cas, $f_i - s_i \neq C_i$.

Au contraire, dans un ordonnancement non préemptif, toute tâche qui débute son exécution la termine sans pouvoir être interrompue. Suivant les cas, la préemption* est autorisée ou non et il existe des algorithmes adaptés pour chacun de ces cas. Dans le cas préemptif, les algorithmes d'ordonnancement font souvent l'hypothèse que les temps de préemption sont nuls. Sous cette hypothèse, autoriser la préemption permet à des jeux de tâches non ordonnançables dans le cas non préemptif de devenir ordonnançables. Cette hypothèse revient néanmoins à négliger les sauvegardes et restaurations de contexte dont le surcoût peut, lors de l'exécution, remettre en question

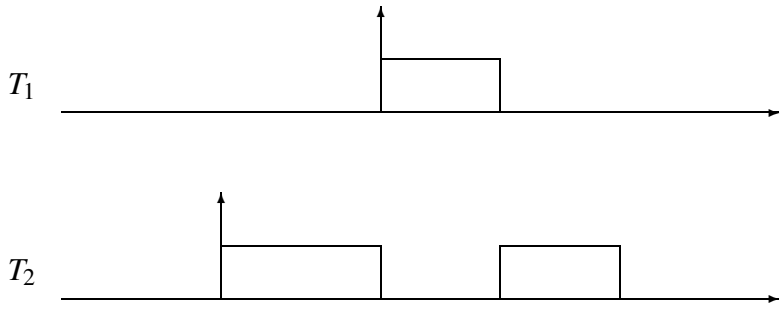

FIG. 6.2 – *Preémption d'une tâche T_2 par T_1*

l'ordonnançabilité théorique. Toutefois, le problème de décider de l'ordonnançabilité d'un jeu de tâches périodiques sans préemption mais avec phase imposée (nulle ou non) a été prouvé NP-difficile [92] et il n'existe pas d'algorithme optimal. Dans la suite de cet état de l'art nous donnons des résultats dans le cas préemptif, sauf si le contraire est précisé.

Dans un ordonnancement, on peut observer parfois une séquence qui se répète. La durée séparant deux répétitions de ce motif est appelée hyperpériode*. Dans le cas d'activation synchrone, c'est-à-dire $\forall i, \Phi_i = 0$, toutes les tâches sont actives à $t = 0$ et le seront à nouveau ensemble à $t = Hp$ où Hp désigne l'hyperpériode qui correspond au plus petit commun multiple (PPCM) des périodes. Dans le cas où les phases sont non nulles, c'est-à-dire $\exists i\, \Phi_i \neq 0$, l'hyperpériode est toujours de la taille du PPCM des périodes, noté P, mais est précédée par un régime transitoire sur l'intervalle $[0, \max(\Phi_i) + P]$. L'hyperpériode correspond alors à l'intervalle $[\max(\Phi_i) + P, \max(\Phi_i) + 2P]$ [96][97].

Il existe deux grandes classes d'algorithmes, les algorithmes à priorités fixes* et les algorithmes à priorités dynamiques*. Un algorithme à priorités fixes consiste à allouer une priorité à chaque tâche. Cela implique que toutes les instances d'une même tâche reçoivent la même priorité (fixe). Au contraire, un algorithme à priorités dynamiques alloue des priorités qui peuvent varier, pour une même tâche, d'une instance à l'autre et/ou au cours du temps pour une même instance. Construire un ordonnancement à l'aide de tels algorithmes consiste à allouer, lors de chaque unité de temps indivisible, le processeur à l'instance de la tâche active la plus prioritaire.

Nous allons présenter les principaux résultats concernant les algorithmes à priorités fixes et à priorités dynamiques dans le cas préemptif. A chaque fois, nous aborderons d'abord le cas particulier où $\forall i, D_i = P_i$, hypothèse fréquente, avant d'élargir au cas où $\forall i, D_i \leq P_i$.

6.1.3.1 Algorithmes à priorités fixes

L'algorithme d'ordonnancement de tâche périodique le plus connu est Rate Monotonic* [98]. Souvent cité comme fondateur de la problématique d'ordonnancement temps réel, cet article décrit l'algorithme Rate Monotonic dans le cas où $\forall i, D_i = P_i$, et démontre son optimalité dans la classe des algorithmes à priorités fixes. Rate Monotonic alloue des priorités inversement proportionnelles aux périodes. La tâche la plus (respectivement la moins) prioritaire est alors celle ayant la période

la plus courte (respectivement la plus longue).

Dans ce même article les auteurs donnent une condition suffisante d'ordonnançabilité avec Rate Monotonic :

THÉORÈME 6.1 (LIU ET LAYLAND)
Soit un jeu de n tâches périodiques avec échéances égales aux périodes. Ce jeu de tâche est ordonnançable avec l'algorithme Rate Monotonic si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \tag{6.1}$$

Le terme de droite est appelé facteur d'utilisation processeur* et représente la fraction du temps processeur consacrée à l'exécution des tâches périodiques. Lorsque n tend vers l'infini, cette borne d'ordonnançabilité tend vers 0,69. Toutefois, dans [99] les auteurs montrent que le facteur moyen d'utilisation processeur atteignable avec des jeux constitués d'un grand nombre de tâches est d'environ 0.88. Récemment une nouvelle condition suffisante d'ordonnançabilité a été présentée dans [100]. Elle donne une borne d'ordonnançabilité moins pessimiste :

THÉORÈME 6.2 (BINI, BUTTAZZO ET BUTTAZZO)
Soit un jeu de n tâches périodiques avec échéances égales aux périodes. Ce jeu de tâches est ordonnançable avec Rate Monotonic si :

$$\prod_{i=1}^{n} \left(\frac{C_i}{P_i} + 1 \right) \leq 2 \tag{6.2}$$

Dans le cas où $\forall i, D_i = P_i$ n'est plus vérifié mais où $\forall i, D_i \leq P_i$, l'algorithme optimal est Deadline Monotonic* [97]. Deadline Monotonic alloue des priorités inversement proportionnelles aux échéances relatives. Il est à noter que Rate Monotonic n'est en fait qu'un cas particulier de Deadline Monotonic (quand $\forall i, D_i = P_i$). Une condition suffisante d'ordonnançabilité d'un jeu de tâches avec Deadline Monotonic peut être obtenu en substituant les échéances relatives aux périodes dans le théorème de Liu et Layland :

THÉORÈME 6.3
Soit un jeu de n tâches périodiques avec échéances inférieures ou égales aux périodes. Ce jeu de tâches est ordonnançable avec l'algorithme Deadline Monotonic si :

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1) \tag{6.3}$$

Néanmoins cette condition d'ordonnançabilité surestime le facteur d'utilisation processeur et reste donc pessimiste.

Dans [101], les auteurs donnent une condition nécessaire et suffisante à l'ordonnançabilité de Deadline Monotonic. Celle-ci fonctionne pour Deadline Monotonic, mais aussi Rate Monotonic ainsi que tout autre algorithme d'ordonnancement à priorités fixes. Cette condition repose sur le calcul du pire temps de réponse* des tâches périodiques, c'est-à-dire pour chaque tâche, la durée

maximum pouvant séparer l'activation d'une instance de sa date de fin d'exécution. En effet, à la durée d'exécution d'une tâche, peut s'ajouter l'occupation du processeur par des tâches plus prioritaires. La tâche peut ainsi être préemptée, ce qui rallonge d'autant son temps de réponse. Si pour une tâche T_i ce pire temps de réponse, noté R_i, est inférieur à son échéance relative, cela signifie qu'elle ne manquera jamais son échéance. Ce qui conduit à la condition nécessaire et suffisante suivante :

THÉORÈME 6.4 (AUDSLEY ET ALL)
Un jeu de tâches périodiques aux échéances inférieures ou égales aux périodes est ordonnançable avec Deadline Monotonic (et donc avec Rate Monotonic dans le cas $\forall i, D_i = P_i$) si et seulement si :

$$\forall i, \quad R_i \leq D_i \tag{6.4}$$

Les auteurs expliquent aussi comment calculer ces pires temps de réponse. Soit un jeu de tâches ordonné par ordre décroissant de priorité (si $i < j$ T_i est plus prioritaire que T_j), le pire temps de réponse d'une tâche T_i, noté R_i est le point fixe solution de l'équation suivante :

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{P_i} \right\rceil C_j \tag{6.5}$$

Ce point fixe est calculé en prenant comme valeur initiale $R_i = C_i$ et en itérant jusqu'à avoir égalité ou que $R_i > D_i$.

6.1.3.2 Algorithmes à priorités dynamiques

Earliest Deadline First* a été utilisé pour la première fois pour des tâches périodiques par Liu et Layland dans [98] avec l'hypothèse $\forall i, D_i = P_i$. Les auteurs démontrent son optimalité dans le cas préemptif et ce pour toutes les classes d'algorithmes. Earliest Deadline First alloue à chaque instant la priorité la plus élevée à la tâche ayant l'échéance absolue la plus proche. Dans ce même article une condition nécessaire et suffisante est donnée.

THÉORÈME 6.5 (LIU ET LAYLAND)
Un jeu de tâches périodiques avec échéances égales aux périodes est ordonnançable avec Earliest Deadline First si et seulement si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{6.6}$$

Cette condition d'ordonnançabilité suffit à elle seule à prouver l'optimalité puisqu'un facteur d'utilisation supérieur à 1 signifierait que la demande de temps processeur est supérieure au temps disponible, quelle que soit la durée considérée.

Il est à noter que plus tard, Mok présentera un autre algorithme d'ordonnancement optimal, Least Laxity First* [102]. Cet algorithme à priorités dynamiques alloue la priorité la plus élevée à la tâche active dont la laxité est la plus petite, la laxité* de l'instance j d'une tâche T_i, notée $X_{i,j}$ étant donnée à une date t par la formule :

$$X_{i,j} = d_{i,j} - t - C_i(t) \tag{6.7}$$

où $C_i(t)$ est la durée d'exécution restante à la date t. Cet algorithme est peu utilisé car il nécessite le calcul des laxités des tâches actives à chaque unité de temps et entraîne davantage de préemptions.

Dans le cas où $\forall i, \quad D_i \leq P_i$, Earliest Deadline First est toujours optimal. Il existe alors une autre condition nécessaire et suffisante [103] plus coûteuse.

THÉORÈME 6.6 (BARUAH, ROSIER ET HOWELL)
Soit $\mathcal{D} = \{d_{i,k} \mid d_{i,k} \leq Hp, 1 \leq i \leq n, k \geq 0\}$, alors un jeu de tâches périodiques avec des échéances relatives inférieures au périodes est ordonnançable avec Earliest Deadline First si et seulement si :

$$\forall L \in \mathcal{D} \quad L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{P_i} \right\rfloor + 1 \right) C_i \tag{6.8}$$

6.1.3.3 Choix d'un algorithme

Il apparaît donc que Earliest Deadline First est optimal et surpasse Deadline Monotonic, que la condition $\forall i, D_i = P_i$ soit vérifiée ou non [1]. On peut s'interroger sur le fait que les algorithmes à priorités fixes soient dans la pratique les plus utilisés. En effet bien que Rate Monotonic et Earliest Deadline First aient été présentés dans le même article, ils ont ensuite donné lieu à des travaux séparés. Des techniques, que nous ne détaillerons pas ici, ont été développées pour ces deux algorithmes afin de gérer, par exemple, les ressources partagées. Ce phénomène se retrouve aussi dans les techniques d'exécution de tâches périodiques et apériodiques. En réalité, l'utilisation par les industriels de Rate Monotonic plutôt qu'Earliest Deadline First repose sur le fait que la plupart des systèmes d'exploitation temps réel ne proposent qu'un ordonnanceur à priorités fixes. Bien qu'un ordonnanceur de type Earliest Deadline First ne soit pas forcément plus coûteux s'il est implémenté dans les services de base du système d'exploitation, son implémentation dans l'application plutôt que dans le système d'exploitation entraîne un surcoût qui nuit à son efficacité. Rarement avancé cet argument est le seul valable. Car la popularité de Rate Monotonic tient aussi à des idées reçues fausses comme le démontre Buttazzo dans [104]. En effet, Earliest Deadline First n'engendre pas plus de préemption, n'entraîne pas de surcoût significatif par rapport à Rate Monotonic lors de l'exécution et permet les mêmes extensions.

6.1.4 Ordonnancement de tâches apériodiques

Le problème d'ordonnancement des tâches apériodiques fermes ou souples a fait ces dernières décennies l'objet de peu de travaux. Le problème d'ordonnancement conjoint de tâches apériodiques et périodiques fut davantage étudié car correspond aux problématiques industrielles mêlant traitement répétitifs et occasionnels. Le problème d'ordonnancement des tâches apériodiques souples seules n'a que peu d'intérêt puisque, en absence de contraintes temps réel, n'importe quel ordre d'exécution convient.

Ainsi, une grande partie des résultats intéressants sur l'ordonnancement des tâches apériodiques sont plus anciens que le terme apériodique lui même. Un premier résultat montre que pour

1. On ne s'intéresse pas ici au cas $D_i > P_i$, car nous considérons que l'instance d'une tâche doit être exécutée avant l'activation de la prochaine instance de cette même tâche.

un jeu de tâches apériodiques fermes activées en même temps, l'algorithme qui les ordonnance par ordre croissant d'échéance absolue est optimal. Cet algorithme est appelé Earliest Due Date [88]. Il est optimal dans le sens où il minimise Re_{max} le retard maximum (*tardiness*) :

$$Re_{max} = \max_{1 \leq i \leq n} (f_i - d_i) \tag{6.9}$$

Si $Re_{max} \leq 0$, l'ensemble des échéances est respectée. Pour décider de l'ordonnançabilité du jeu de tâches, il suffit de vérifier que $\forall i,\ f_i \leq d_i$ ce qui revient à, si les tâches sont indicées par ordre croissant d'échéance absolue :

$$\forall i = 1, \ldots, n \quad \sum_{k=1}^{i} C_k \leq d_i \tag{6.10}$$

Si les tâches apériodiques fermes sont maintenant à dates d'activation quelconques, Horn [89] a prouvé que l'algorithme qui à chaque instant exécute la tâche active à l'échéance absolue la plus proche minimise Re_{max}. A un instant t, si une tâche est activée, on doit décider de la rejeter ou garantir qu'elle sera exécutée avant son échéance sans qu'aucune autre tâche préalablement acceptée ne manque la sienne. Pour cela on considère l'ensemble des tâches apériodiques préalablement acceptées auquel on ajoute la nouvelle tâche. Puis on ordonne cet ensemble de tâches par ordre croissant d'échéance absolue, J_1 étant ainsi la tâche ayant l'échéance la plus proche et J_n celle dont l'échéance est la plus tardive. Soit $C_{restant_i}(t)$ la durée d'exécution restante pour la tâche apériodique J_i à la date t. La nouvelle tâche peut être acceptée si :

$$\forall i = 1, \ldots, n \quad \sum_{k=1}^{i} C_{restant_k}(t) \leq d_i \tag{6.11}$$

Un autre résultat intéressant concerne l'ordonnançabilité d'un jeu de tâches apériodiques fermes avec contraintes de précédence dans le cas où les dates d'activation sont connues. Dans [105] les auteurs expliquent comment transformer un jeu de tâches $\mathcal{J}$ avec contraintes de précédence en un jeu de tâches $\mathcal{J}^*$ sans contraintes de précédence. Le nouveau jeu de tâches est ensuite ordonnancé avec l'algorithme Earliest Deadline First. Les auteurs démontrent ensuite que $\mathcal{J}$ est ordonnançable si et seulement si $\mathcal{J}^*$ est ordonnançable. La transformation consiste en fait à modifier les dates d'activation et les échéances de manière à ce que chaque tâche ne puisse pas commencer avant la fin de ses prédécesseurs et soit exécutée avant ses successeurs.

Dans [106], les auteurs s'intéressent à l'ordonnancement de tâches apériodiques fermes à activations inconnues. Les applications industrielles comme les serveurs web ou les routeurs nécessitent, lors de l'exécution et à faible surcoût, d'accepter ou de rejeter des tâches apériodiques fermes. Le but est alors de trouver une condition suffisante d'ordonnançabilité. Pour obtenir leurs résultats, ils prouvent d'abord l'équivalence du modèle de tâche classique avec leur modèle de tâche acyclique (*acyclic task model*). Ainsi on peut utiliser indifféremment l'un ou l'autre des modèles pour statuer sur l'ordonnançabilité. Ensuite, ils prouvent que Deadline Monotonic est optimal pour l'ordonnancement de tâches apériodiques fermes dans la classe des algorithmes à priorités fixes. On note $S(t)$ l'ensemble contenant la nouvelle tâche et les tâches apériodiques fermes préalablement acceptées et dont l'activation a eu lieu après la dernière période d'inactivité processeur.

Le théorème suivant permet d'accepter ou de rejeter cette nouvelle tâche :

THÉORÈME 6.7 (ABDELZAHER, SHARMA ET LU)
Connaissant l'ensemble des tâches préalablement acceptées, accepter la nouvelle tâche en garantissant le respect de l'ensemble des échéances est possible si

$$\sum_{T_i \in S(t)} C_i/D_i \leq \frac{1}{1+\sqrt{1/2}} \tag{6.12}$$

Cette borne est environ égale à 0,586.

6.1.5 Ordonnancement de tâches sporadiques

Le problème d'ordonnancement de tâches sporadiques est plus proche de celui des tâches périodiques qu'apériodiques. En effet, comme pour les tâches périodiques, il s'agit de vérifier avant exécution, l'ordonnançabilité d'un jeu de tâches. Lorsque tâches sporadiques et périodiques sont ordonnancées ensemble, les premières sont souvent transformées en tâches périodiques. La méthode la plus simple est appelée *polling*. Elle consiste à créer une tâche périodique pour chaque tâche sporadique. Lorsqu'une tâche de ce type est exécutée, elle donne son temps d'exécution à la tâche sporadique associée s'il en existe une instance active. Le cas échéant elle termine immédiatement son exécution. Cette méthode est fréquemment utilisée dans l'industrie pour exécuter des traitements se répétant de manière irrégulière mais sujets à des contraintes temps réel strictes comme le freinage pour une voiture par exemple.

Les travaux consacrés uniquement à l'ordonnancement de tâches sporadiques sont rares. Certains résultats viennent de travaux sur l'ordonnancement mixte de tâches périodiques et apériodiques qui sont ensuite adaptés au cas sporadique. C'est le cas par exemple du serveur sporadique [93] initialement développé pour l'ordonnancement de tâches apériodiques souples avec tâches périodiques. Les auteurs indiquent que pour chaque tâche sporadique, une tâche supplémentaire, appelée serveur, peut être allouée. Ils montrent notamment que si un jeu de tâches périodiques contenant une tâche T_i est ordonnançable, il est aussi ordonnançable si T_i est remplacée par un serveur sporadique avec la même période et la même durée. Les conditions d'ordonnançabilité restent ainsi inchangées.

Dans [92] le cas non préemptif pour des tâches sporadiques est abordé. Dans ce cas Earliest Deadline First est prouvé optimal et une condition nécessaire d'ordonnançabilité est présentée.

Il existe d'autres résultats mais appliqués à des modèles de tâches différents du modèle traditionnel. Dans le *multiframe task model* de Mok et Chen [91] les tâches sont toutes sporadiques. La nouveauté vient du fait que les tâches ont une durée d'exécution variable mais prenant des valeurs connues. La tâche $T_i((1,3),4)$ est par exemple une tâche de durée minimum d'inter-activations de 4 et dont les instances ont successivement pour durée 1 puis 3. Les auteurs montrent que Rate Monotonic reste optimal pour ce modèle et donnent une condition suffisante d'ordonnançabilité sous forme de borne d'ordonnançabilité.

Baruah introduit dans [107] un modèle encore différent. Dans ce modèle, une tâche est représentée par un graphe orienté de sous-tâches avec une seule sous-tâche d'entrée ainsi qu'une

seule sous-tâche de sortie. Chaque sous-tâche possède une durée d'exécution et une échéance. Chaque arc indique une durée minimum séparant l'activation de la sous-tâche source de la sous-tâche destination. Chaque tâche (graphe) possède en outre une durée minimum d'inter-activations s'appliquant à l'unique sous-tâche d'entrée.

Une fois ce modèle présenté, l'article explique comment vérifier qu'un ensemble de tâches Γ est ordonnançable en vérifiant :

$$\sum_{T \in \Gamma} dbf(T,t) \leq t, \forall t \geq 0 \tag{6.13}$$

dbf est la fonction qui renvoie la demande (temps d'exécution) cumulée de toutes les sous-tâches d'une tâche T pouvant être activées et possédant leur échéance sur une durée t. Un algorithme pour calculer dbf est donné, puis il est indiqué comment restreindre le test à $t \in [0,lmt]$ où lmt est un entier qu'une méthode permet de calculer.

Ces deux modèles ont donné lieu à un troisième modèle, le *generalized multiframe task model* [108] qui reprend les principes du *multiframe task model* mais où chaque instance d'une tâche peut, en plus d'avoir une durée différente, avoir une échéance différente de la durée minimum d'inter-activations. De plus cette échéance peut varier, au même titre que la durée minimum d'inter-activations comme dans les sous-tâches du modèle de Baruah. Une condition d'ordonnançabilité est donnée pour ce modèle, sous quelques restrictions.

6.2 Hors-ligne et en-ligne

On différencie les approches hors-ligne* (*off-line*) des approches en-ligne* (*on-line*). Hors-ligne signifie avant exécution et s'oppose à en-ligne, c'est-à-dire durant l'exécution. Cette opposition concerne l'implémentation de l'ordonnancement ou de l'algorithme d'ordonnancement. Dans une approche en-ligne, c'est une tâche supplémentaire appelée ordonnanceur* (*scheduler*) qui choisit à chaque instant quelle tâche exécuter. La durée d'exécution de l'ordonnanceur est habituellement négligée. Pour décider quelle tâche doit être exécutée, un ordonnanceur peut utiliser des priorités spécifiées par l'utilisateur, ou suivre les principes d'un algorithme d'ordonnancement comme Rate Monotonic ou Earliest Deadline First. On parle alors d'ordonnancement en-ligne.

Mais rien n'empêche, connaissant un jeu de tâches, de tester son ordonnançabilité avec un algorithme d'ordonnancement et, grâce à cet algorithme, de déduire un ordre et des dates d'exécution qui satisfassent les contraintes temps réel. Cet ordre et ces dates d'exécution générés hors-ligne peuvent alors être stockées en mémoire. Cela va même jusqu'à inclure et fixer les dates de préemption. Ainsi dans une approche hors-ligne l'ordonnancement calculé avant exécution est "lu" par une tâche supplémentaire qui, contrairement à l'ordonnanceur ne fait pas de choix. On parle alors d'ordonnancement hors-ligne.

Il est important de noter qu'un même algorithme d'ordonnancement peut être utilisé selon les deux approches, soit en l'implémentant sous forme d'un ordonnanceur pour l'approche en-ligne, soit en générant hors-ligne les dates d'exécution des tâches et en les stockant dans une table pour l'approche hors-ligne. On ne peut donc qualifier un algorithme comme Rate Monotonic ou Earliest Deadline First de hors-ligne ou de en-ligne.

Un premier intérêt de l'approche hors-ligne est que le surcoût dû à l'ordonnancement lors de l'exécution est plus faible que dans une approche en-ligne. De plus ce surcoût peut être estimé précisément et être pris en compte dans les algorithmes d'ordonnancement eux-mêmes. On peut ainsi avoir une implémentation totalement déterministe, ce qui est nécessaire pour certaines applications qui nécessitent certifications et preuves comme c'est par exemple le cas dans l'aéronautique.

Un deuxième intérêt de l'approche hors-ligne est que les dates d'exécution, puisqu'elles sont connues, peuvent être prises en compte dans les applications. Lorsqu'un jeu de tâches constitue une loi de commande par exemple, cette dernière est sensible aux variations des durées séparant les tâches d'acquisitions, de traitements et de réactions. On pourra choisir une approche hors-ligne de manière à garantir des durées fixes, que se soit entre deux instances d'une tâche d'acquisitions ou entre une tâche d'acquisition et une tâche de réaction. Il existe aussi des résultats d'automatique permettant, si on connaît les dates d'exécution de la chaîne acquisition-traitement-réaction, de corriger ponctuellement les lois de commandes pour s'adapter à ces variations [109].

Un troisième intérêt est que dans les approches hors-ligne la complexité des algorithmes d'ordonnancement n'a pas d'influence sur l'exécution temps réel. Le seul surcoût est celui de la lecture de la table d'ordonnancement. En revanche les approches en-ligne nécessitent que le temps d'exécution de l'ordonnanceur reste négligeable, ce qui implique que l'algorithme d'ordonnancement soit de faible complexité. Un tel algorithme ne peut être efficace que si le jeu de tâches satisfait des hypothèses simplificatrices assez restrictives, comme les échéances égales aux périodes ou l'absence de ressources partagées. En cas de ressources partagées, de contraintes de précédence ou autres, la complexité des algorithmes augmente. Si on considère le cas distribué, où différents processeurs sont reliés par différents médias de communication, il n'existe plus d'algorithme optimal. Trouver un ordonnancement ainsi qu'une distribution, nécessite alors d'employer des heuristiques. Ces heuristiques de part leur complexité, ne peuvent être employées en-ligne. De plus elles servent souvent à dimensionner l'architecture distribuée. Ainsi, une approche hors-ligne est souhaitée dès qu'il y a distribution [3], d'autant plus si les communications et leurs coûts sont pris en compte [2][67].

Néanmoins, l'approche hors-ligne a elle aussi ses limites. On ne peut en effet pas maîtriser les comportements non prévus à l'avance. L'ordonnancement de tâches apériodiques par exemple ne peut se faire qu'en-ligne. De même, le rajout d'une tâche supplémentaire exige de recalculer et de remplacer la table des dates d'exécution alors que dans une approche en-ligne il suffit, après étude d'ordonnançabilité, d'ajouter la tâches avec ses caractéristiques.

Il est aussi souvent reproché aux approches hors-ligne leur coût mémoire. En effet, il faut stocker en mémoire l'ordre et les dates d'exécution des tâches sur l'hyperpériode (motif se répétant). Si les périodes sont quelconques, cette hyperpériode peut être très grande et la quantité mémoire nécessaire augmenter en conséquence. Néanmoins, si les périodes ne sont pas toutes harmoniques entre elles, elles sont souvent multiples entre elles. En outre, c'est dans l'intérêt des développeurs de choisir de telles périodes car cela augmente l'ordonnançabilité du système. Dans [110], les auteurs donnent une méthode pour modifier les périodes de manière à les rendre multiples en elles et montrent l'impact que cela a sur l'ordonnançabilité.

Il est bon d'ajouter qu'il est possible de passer d'un ordonnancement hors-ligne à un ordonnancement en-ligne. On peut par exemple produire un ordonnancement hors-ligne avec des heu-

ristiques, puis transformer cet ordre et ces dates d'exécution en caractéristiques de tâches pour un ordonnanceur à priorités fixes [111] et donc faire de l'ordonnancement en-ligne. Cette transformation peut aussi avoir lieu pour un ordonnanceur reposant sur Earliest Deadline First [112]. L'intérêt du jeu de tâches obtenu est qu'il n'y a jamais deux tâches actives en même temps (sauf si des tâches apériodiques sont rajoutées) et qu'ainsi le surcoût dû à l'ordonnanceur est très faible.

6.3 Ordonnancement mixte de tâches périodiques et apériodiques

Le problème auquel on s'intéresse ici est l'ordonnancement mixte de tâches périodiques et apériodiques. Ce problème sous-entend qu'un premier problème a été résolu, à savoir l'ordonnancement des tâches périodiques. En fonction du jeu de tâches périodiques, on a choisi un algorithme d'ordonnancement avec lequel on vérifie l'ordonnançabilité du jeu de tâches. On veut alors, en-ligne, exécuter des tâches apériodiques fermes ou souples, tout en respectant les contraintes des tâches périodiques. On considère que la partie périodique du système est temps réel strict et que la partie apériodique est temps réel mou.

Dans le cas des tâches apériodiques souples, on s'attachera à minimiser le temps de réponse (défini au 6.1.3.1 page 142) ou, à défaut, le temps de réponse moyen. On qualifiera donc un algorithme d'optimal* s'il permet de satisfaire les contraintes des tâches périodiques et minimise le temps de réponse de chaque tâche apériodique lorsque les tâches apériodiques souples sont exécutées dans l'ordre de leur activation.

Dans le cas des tâches apériodiques fermes le temps de réponse n'a plus d'importance, car seul le respect des échéances est important. Le critère d'optimalité* d'un algorithme permettant de satisfaire les contraintes des tâches périodiques et d'exécuter des tâches apériodiques fermes est alors difficile à définir.

La méthode la plus simple à implémenter afin d'ordonnancer des tâches périodiques et apériodiques est l'ordonnancement en tâche de fond* (*background scheduling*). Avec cette méthode les tâches périodiques sont prioritaires sur les tâches apériodiques. Ces dernières ne peuvent s'exécuter que lorsqu'aucune tâche périodique n'est active.

La figure 6.3 page 150 donne un exemple d'ordonnancement en tâche de fond. Deux tâches périodiques T_1 et T_2 ont respectivement pour durée d'exécution 6 et 1 et possèdent chacune une échéance relative égale à leur période valant respectivement 12 et 4. Ces deux tâches sont ordonnancées avec Rate Monotonic, la tâche T_2 étant ainsi plus prioritaire que T_1. Leur ordonnancement n'est pas modifié par les tâches apériodiques et est représenté sur les deux diagrammes supérieurs de la figure. Imaginons que l'on veuille faire de l'ordonnancement en tâche de fond de tâches apériodiques souples en plus de ces deux tâches périodiques. Le troisième diagramme de la figure montre les dates d'activation et l'exécution des ces tâches apériodiques souples. On peut ainsi remarquer que la tâche apériodique souple J_{aps2} de durée 1 et survenant à $t = 10$ peut être exécutée aussitôt, aucune tâche périodique n'étant active. Par contre, la tâche J_{aps1} de durée 1 qui survient à $t = 3$ ne peut être exécutée avant $t = 9$. On comprend facilement que les temps de réponse vont dépendre grandement de la répartition des temps d'inactivité processeur. Dans le cas des tâches

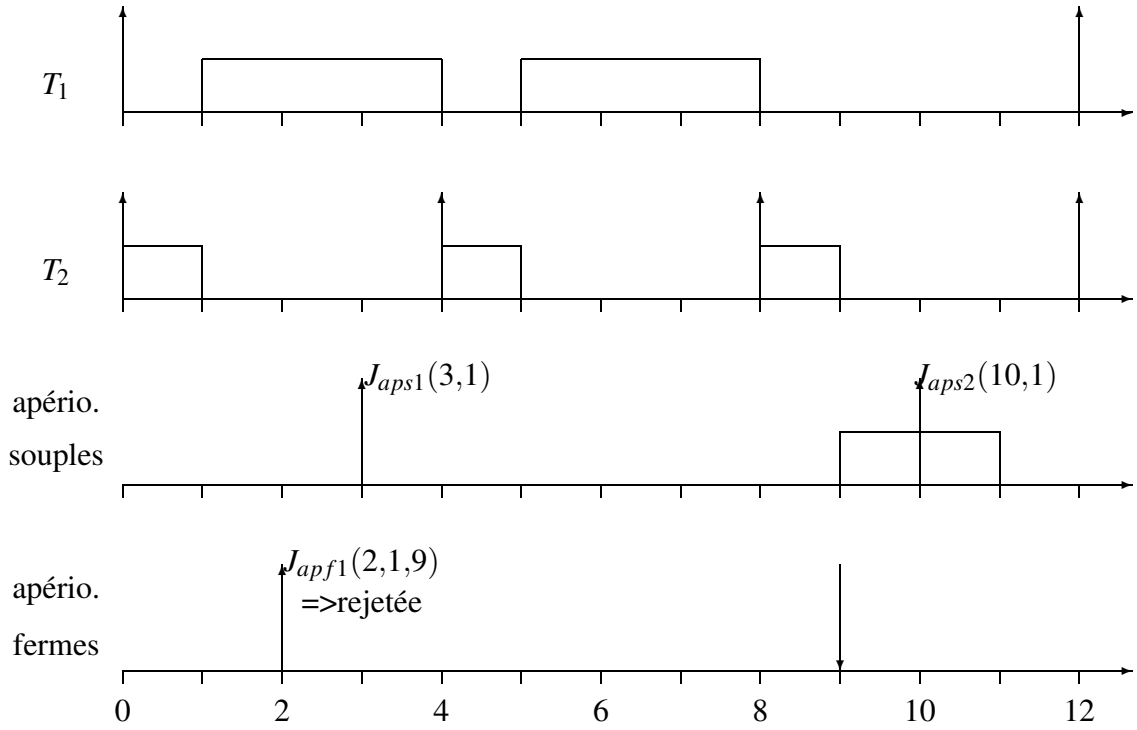

FIG. 6.3 – *Méthode d'ordonnancement en tâche de fond*

apériodiques fermes, le quatrième diagramme montre qu'une tâche activée à $t = 2$, de durée 1 et d'échéance absolue 9 est rejetée. En effet, le processeur est occupé durant tout l'intervalle [2,9]. Pourtant, on peut imaginer qu'exécuter cette tâche dans l'intervalle [2,3] repousserait la fin d'exécution de T_1 à $t = 10$ mais ne l'empêcherait pas de satisfaire son échéance. En conclusion, s'il est facile d'implémentation, l'ordonnancement en tâche de fond ne permet pas d'obtenir de bons résultats, aussi bien au niveau du temps de réponse que de l'acceptation de tâches apériodiques fermes. La plupart des travaux mentionnent néanmoins l'ordonnancement en tâche de fond à titre de comparaison.

Les travaux que nous présentons dans la suite de cet état de l'art concernent principalement les tâches apériodiques souples. Bien que certains puissent être étendus aux tâches apériodiques fermes, ils n'y sont pas forcément adaptés. Nous préciserons pour chacun le type de tâche apériodique visé et les éventuelles extensions. Enfin, les approches présentées dépendent pour la plupart de l'algorithme d'ordonnancement utilisé pour les tâches périodiques. Nous classerons donc ces approches selon qu'elles reposent sur un ordonnancement des tâches périodiques par un algorithme à priorités fixes, par un algorithme à priorités dynamiques, ou qu'elles sont indépendantes de l'algorithme employé.

6.3.1 Approches à priorités fixes

Sont ici détaillées les techniques permettant d'ordonnancer des tâches périodiques avec des tâches apériodiques souples ou fermes, en utilisant l'algorithme d'ordonnancement Rate Monotonic pour l'ordonnancement des tâches périodiques, ainsi que les résultats d'ordonnançabilité associés. On se restreint donc au cas pour lequel cet algorithme est optimal, à savoir quand $\forall i, D_i = P_i$ pour l'ensemble des tâches périodiques. Par ailleurs, mais uniquement par soucis de simplification[2], on considère que l'ensemble des tâches périodiques sont actives à $t = 0$, c'est-à-dire que les phases sont nulles. Les tâches périodiques seront donc notées $T_i(C_i,P_i)$, les tâches apériodiques souples $J_i(r_i,C_i)$ et les tâches apériodiques fermes $J_i(r_i,C_i,d_i)$.

6.3.1.1 Serveur à scrutation

Première méthode, le serveur à scrutation* (*polling server*) [113] est une extension de la méthode de polling décrite précédemment qui consiste à périodiquement scruter si un événement a eu lieu ou non. Appliquée à l'ordonnancement mixte de tâches périodiques et apériodiques souples, elle consiste à ajouter une tâche périodique appelée serveur, dont la période est notée P_s et dont la durée d'exécution est appelée capacité* et notée C_s. Lorsque le serveur est exécuté, il donne sa capacité aux tâches apériodiques souples actives. Il existe d'autre types de serveur que le serveur à scrutation, mais tous repose sur ce principe de tâche ajoutée qui donne sa capacité aux tâches apériodiques souples.

Il s'agit donc hors-ligne d'ajouter une tâche au jeu de tâches périodiques existant et pour cela d'utiliser une condition d'ordonnançabilité afin de déterminer C_s et P_s. Ensuite, la manière dont le serveur est utilisé, en-ligne, pour exécuter les tâches apériodiques et comment il retrouve sa capacité initiale, diffère d'un type de serveur à l'autre.

Pour le serveur à scrutation, la capacité est rafraîchie, c'est-à-dire retrouve son niveau initial au début de chaque période. La file d'attente des tâches apériodiques souples peut être ordonnée par différentes politiques comme FIFO (*First In First Out* pour première arrivée, première exécutée) ou par ordre croissant de durée d'exécution restante. Si à son exécution, ou au cours de celle-ci, il n'y a plus aucune tâche apériodique souple active, le serveur cesse son exécution et sa capacité est perdue jusqu'à la prochaine période. Habituellement, on choisit la période du serveur de manière à ce qu'il soit la tâche la plus prioritaire ou, le cas échéant, ait une priorité élevée.

La figure 6.4 page 152 donne un exemple d'ordonnancement avec serveur à scrutation. Le jeu de tâches périodiques est composé de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Le serveur a pour capacité 2 et pour période 6. De ce fait, le serveur est la tâche la plus prioritaire. A $t = 0$, il n'y a pas de tâches apériodiques souples en attente, le serveur n'est donc pas exécuté et sa capacité est perdue. La tâche apériodique souple $J_1(3,2)$ ne peux donc être traitée qu'à $t = 6$ lorsque la capacité du serveur est rafraîchie. Le serveur donne alors la totalité de sa capacité à la tâche en attente. A $t = 12$, le serveur exécute $J_2(9,1)$ puis, en absence d'autre tâche apériodique, termine son exécution et perd la capacité restante. Ainsi, $J_3(15,1)$ doit attendre le prochain rafraîchissement de cette capacité pour être exécutée à $t = 18$.

2. Cette hypothèse peut être levée sans remettre en cause les résultats présentés.

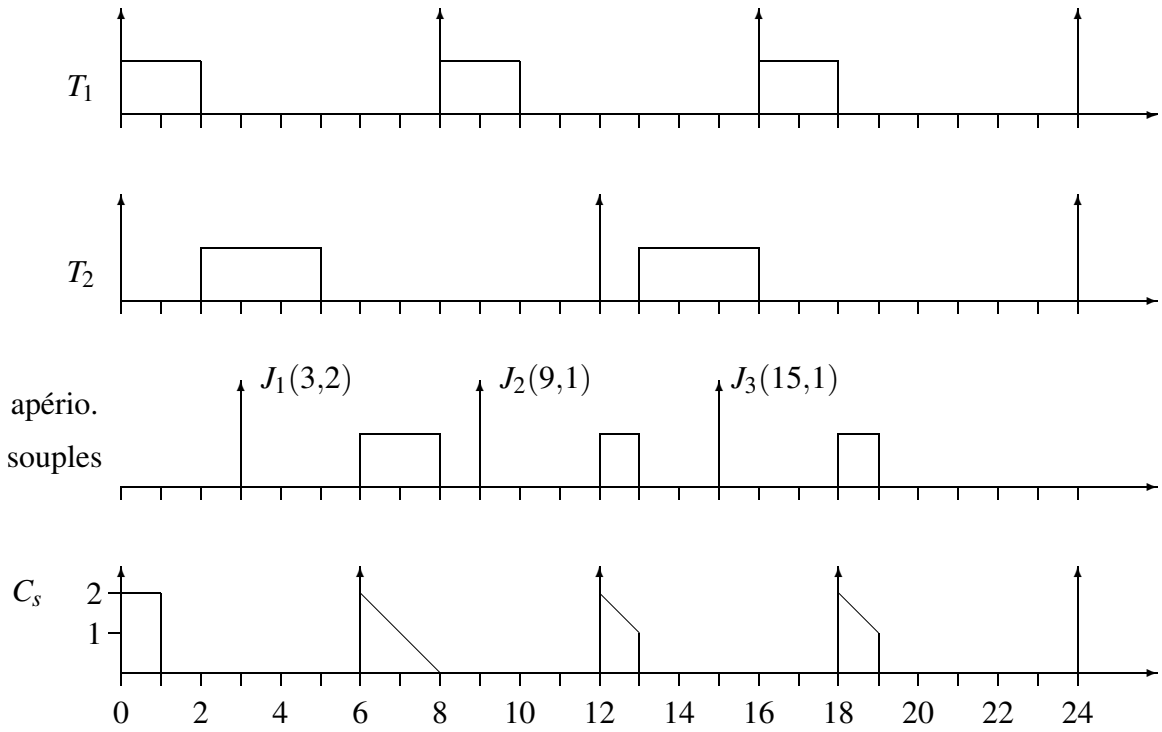

FIG. 6.4 – *Serveur à scrutation*

On comprend bien avec cet exemple que le temps de réponse des tâches apériodiques souples avec un serveur à scrutation va dépendre de leur date d'activation. Si aucune tâche apériodique souple n'est en attente, le serveur se termine immédiatement et sa capacité est perdue. Si une tâche apériodique souple est activée juste après, elle doit attendre l'équivalent d'une période avant d'être traitée. Néanmoins, en ce qui concerne l'ordonnançabilité, le serveur étant une tâche comme les autres, les résultats d'ordonnançabilité vu au 6.1.3.1 page 141 s'appliquent toujours. Ainsi, un jeu de n tâches périodiques auquel est ajouté un serveur de capacité C_s et de période P_s est ordonnançable si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \frac{C_s}{P_s} \leq (n+1)[2^{1/(n+1)} - 1] \tag{6.14}$$

En ce qui concerne les tâches apériodiques fermes, la technique consiste à inclure la nouvelle tâche dans la file d'attente des tâches apériodiques fermes préalablement acceptées, ordonnées par ordre croissant d'échéance absolue. Ensuite, on calcule la date de fin d'exécution de chaque tâche. La nouvelle tâche est acceptée si pour toute tâche de la file $f_i \leq d_i$, sinon elle est rejetée.

6.3.1.2 Serveur ajournable

Le but du serveur ajournable* (*deferrable server*) [113][114] est d'améliorer le temps de réponse moyen des tâches apériodiques souples par rapport à l'ordonnancement en tâche de fond et au serveur à scrutation. Pour cela, on utilise aussi un serveur de capacité C_s et de période P_s, mais

contrairement au serveur à scrutation, la capacité n'est pas perdue lorsqu'il n'y a pas de tâches apériodiques actives. Le serveur peux donc s'exécuter, consacrer une partie de sa capacité à une tâche, se suspendre, puis s'exécuter à nouveau si un autre tâche apériodique souple est activée et qu'il n'y a pas de tâches périodiques plus prioritaires. La capacité est rafraîchie (entièrement disponible) au début de chaque période.

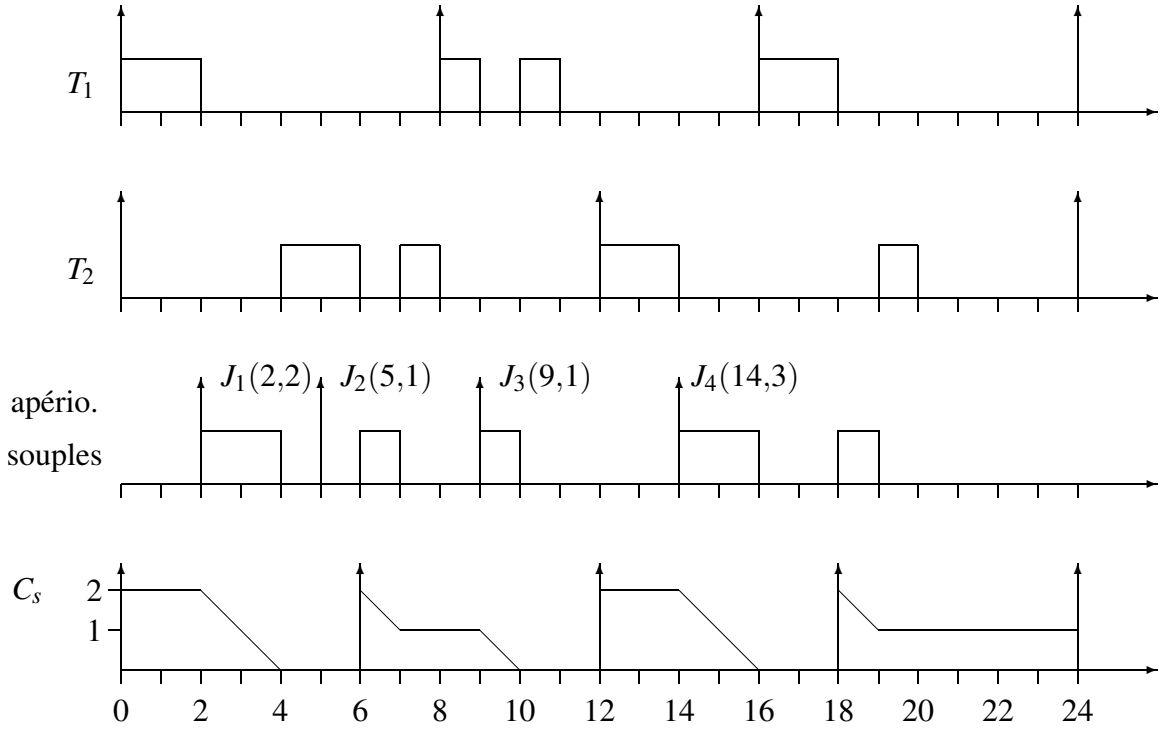

FIG. 6.5 – *Serveur ajournable*

La figure 6.5 page 153 donne un exemple d'ordonnancement avec serveur ajournable. Le jeu de tâches périodiques est composé de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Le serveur a pour capacité 2 et pour période 6, c'est la tâche la plus prioritaire. A $t = 0$, il n'y a pas de tâche apériodique souple active, le serveur se suspend mais ne perd pas sa capacité, si bien que lorsque la tâche apériodique souple $J_1(2,2)$ est activée, le serveur peut lui donner les deux unités de temps de sa capacité. Lorsque $J_2(5,1)$ est activée, la totalité de la capacité du serveur a été consommée et la tâche doit attendre le début de la prochaine période pour être exécutée ($t = 6$). Cette tâche ne consommant pas la totalité de la capacité du serveur, la tâche apériodique $J_3(9,1)$ peut être exécutée immédiatement après son activation. Enfin, la tâche $J_4(14,3)$ nécessitant plus de temps d'exécution que la capacité du serveur, elle est exécutée en deux temps, au cours de deux périodes consécutives.

En comparant les deux exemples 6.4 page 152 et 6.5 page 153, on se rend compte facilement du gain qu'apporte le serveur ajournable au niveau du temps de réponse moyen. Cependant, ce gain se fait au détriment de l'ordonnançabilité. En effet le serveur ajournable peut se suspendre lui-même ce qui est en contradiction avec la définition donnée au 6.1.1 page 135 de ce qu'est une tâche. Contrairement au serveur à scrutation, on ne peut pas prendre un jeu de tâches périodiques

ordonnançable et remplacer une des tâches par le serveur sans en réévaluer l'ordonnançabilité.

THÉORÈME 6.8 (STROSNIDER, LEHOCZKY ET SHA)
Un jeu de n tâches périodiques est ordonnançable avec un serveur ajournable de capacité C_s et de période P_s si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \ln\left(\frac{U_s+2}{2U_s+1}\right) \quad \text{où } U_s = C_s/P_s \tag{6.15}$$

En ce qui concerne les tâches apériodiques fermes, la technique est la même que pour le serveur à scrutation, l'algorithme de calcul des dates de fin d'exécution se compliquant quelque peu.

6.3.1.3 Serveur à échange de priorités

Le serveur à échange de priorités* (*priority exchange*) [113] a pour but d'offrir une borne d'ordonnançabilité supérieure au serveur ajournable tout en approchant ses performances au niveau des temps de réponse. Là encore, une tâche est rajouté avec une capacité C_s et une période P_s. Comme avec le serveur ajournable, la capacité non consommée n'est pas perdue mais la façon dont elle est sauvegardée diffère. Lorsque le serveur n'a pas de tâche apériodique souple à exécuter, il échange sa priorité avec la tâche périodique active la plus prioritaire. Une capacité de durée égale à la durée de l'échange est alors accumulée au niveau de priorité d'origine de la tâche. On peut donc avoir de la capacité à différent niveau de priorité. Lorsqu'une tâche apériodique souple est activée, elle peut s'exécuter s'il y a de la capacité disponible à un niveau de priorité supérieur ou égal à celui de la tâche périodique la plus prioritaire.

La figure 6.6 page 155 donne un exemple d'ordonnancement avec serveur à échange de priorités. Le jeu de tâches périodiques est composé de deux tâches $T_1(5,12)$ et $T_2(10,24)$. Le serveur a une capacité de 1 et une période de 6, ce qui fait de lui la tâche la plus prioritaire. A $t = 0$, en absence de tâche apériodique, le serveur laisse la tâche T_1 s'exécuter et la capacité du serveur est conservée au niveau de priorité de T_1. A $t = 5$, cette capacité change de niveau de priorité car T_1 est terminée et laisse T_2 s'exécuter. Lorsque $J_1(6,1)$ est activée, elle utilise la capacité disponible au plus au niveau de priorité, c'est-à-dire le niveau du serveur dont la capacité vient d'être rafraîchie. La capacité accumulée au niveau de priorité de T_2 est donc conservée. A $t = 12$ le serveur laisse encore une fois T_1 s'exécuter et la capacité est conservée au niveau de priorité de T_1. Cette capacité, plus prioritaire que celle accumulée au niveau de priorité de T_2, est utilisée lorsque $J_2(14,1)$ est activée. A $t = 18$, la capacité du serveur est conservée au niveau de priorité de T_2 car c'est la seule tâche active. La tâche $J_3(20,1)$ consomme une des deux unités de capacité accumulées au niveau de priorité de T_2. Enfin, à $t = 23$, puisqu'il n'y a plus de tâche active, les capacités accumulées se vident au rythme d'une unité de capacité par unité de temps d'inactivité, en commençant par le plus haut niveau de priorité.

Grâce aux échanges de priorités, le serveur simule le comportement d'une tâche normale (ne se suspendant pas), contrairement au serveur ajournable. Du coût, la borne d'ordonnançabilité est grandement améliorée.

THÉORÈME 6.9 (STROSNIDER, LEHOCZKY ET SHA)
Un jeu de n tâches périodiques est ordonnançable avec un serveur à échange de priorités de capacité

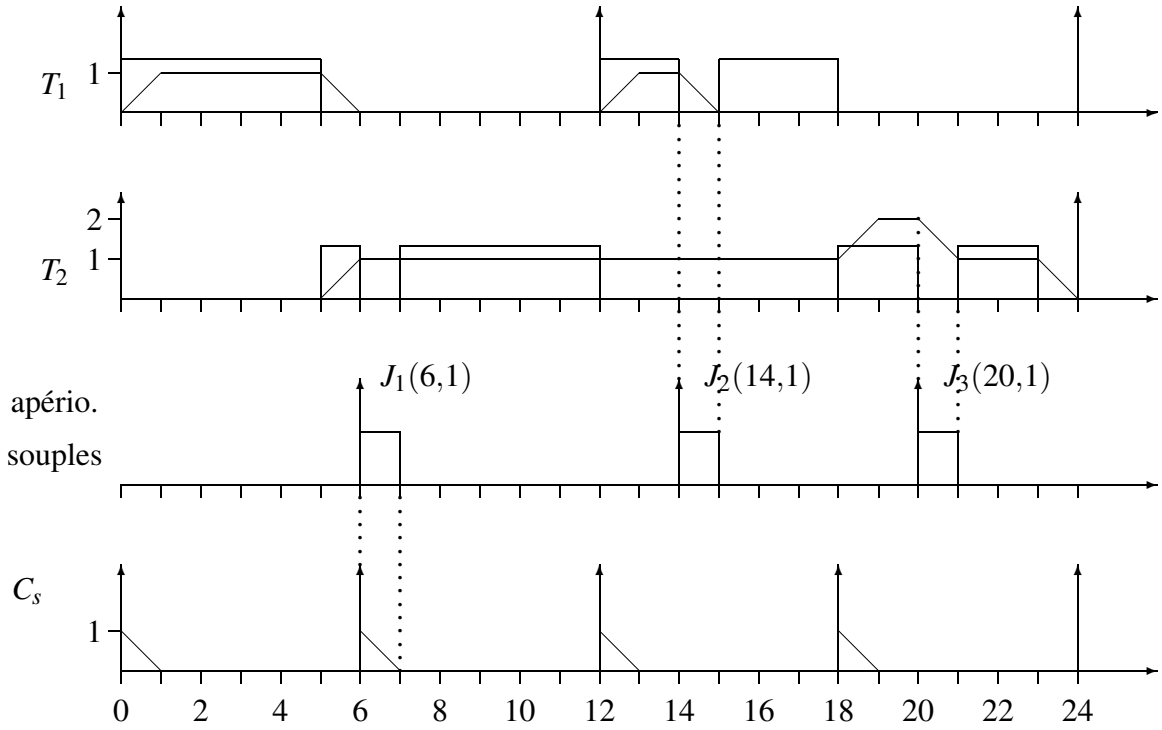

FIG. 6.6 – *Serveur à échange de priorités*

C_s et de période P_s si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \ln\left(\frac{1}{U_s+1}\right) \quad \text{où } U_s = C_s/P_s \tag{6.16}$$

6.3.1.4 Serveur sporadique

Ici, le qualificatif sporadique désigne la manière dont la capacité du serveur est rafraîchie et non le type de tâche qu'il traite. En effet le serveur sporadique* (*sporadic server*) [93] est une autre technique d'ordonnancement mixte de tâches périodiques et apériodiques souples qui permet d'obtenir des temps de réponse plus proches de ceux obtenus avec le serveur ajournable tout en gardant une borne d'ordonnançabilité élevée. Cette technique consiste également à rajouter une tâche, le serveur, avec une capacité C_s et une période P_s. Comme pour le serveur ajournable, la capacité qui n'est pas utilisée n'est pas perdue et se conserve au niveau de priorité du serveur. La différence concerne la façon dont cette capacité est rafraîchie. En effet, ce rafraîchissement n'a plus lieu périodiquement à chaque début de période.

On dit que le serveur est actif si le niveau de priorité de la tâche exécutée est supérieur ou égal au sien et que la capacité est non nulle. On appelle intervalle d'activité du serveur l'intervalle $[t_d,t_f]$ tel qu'à $t = t_d$ le serveur devient actif et le reste jusqu'à $t = t_f$ où soit une tâche moins prioritaire débute son exécution, soit $C_s = 0$. A chaque intervalle d'activité du serveur va correspondre une date de rafraîchissement. Ce rafraîchissement aura lieu à $t = t_d + P_s$ et la capacité sera augmentée

d'autant d'unités de temps qu'il y en a eu de dépensées dans l'intervalle $[t_d,t_f]$.

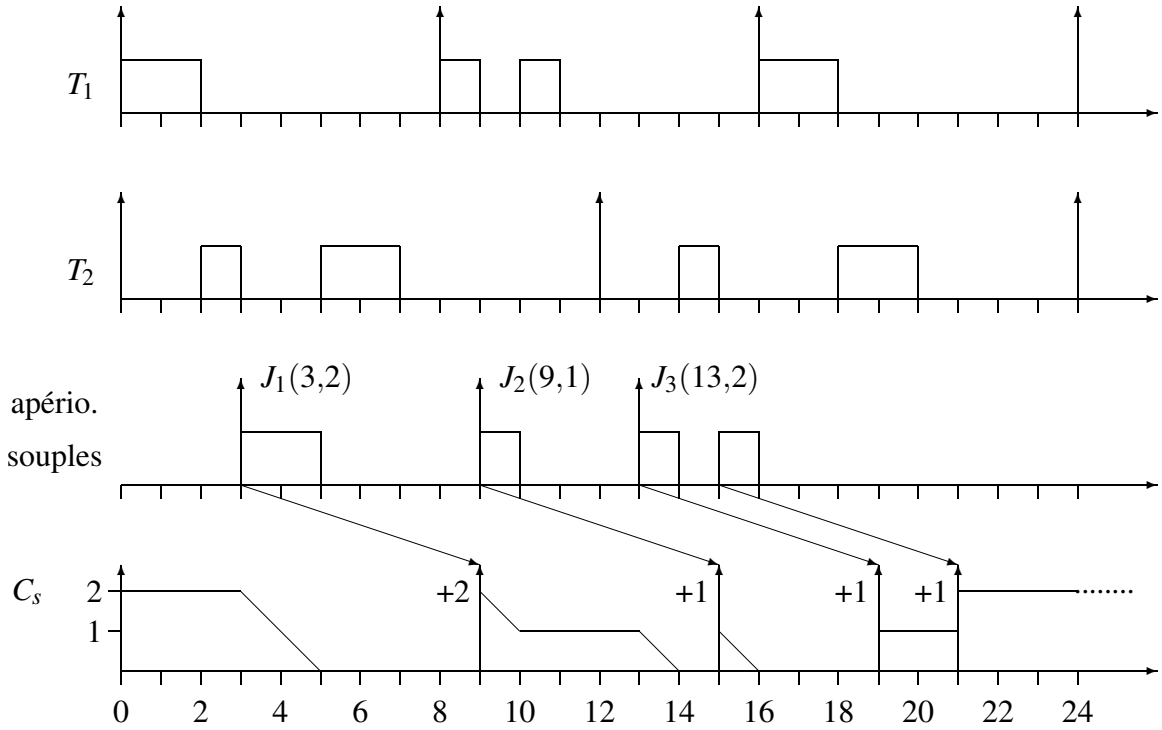

FIG. 6.7 – *Serveur sporadique*

La figure 6.7 page 156 donne un exemple d'ordonnancement avec serveur sporadique. Le jeu de tâches périodiques est constitué de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Le serveur sporadique a une capacité de 2 et une période de rafraîchissement de 6. A $t = 0$ le serveur laisse T_1 s'exécuter mais conserve sa capacité, qu'il peut utiliser à $t = 3$ lorsque $J_1(3,2)$ est activée. Sur cet intervalle d'activité [3,5], le serveur consomme deux unités de sa capacité qui lui sont redonnées à $t = 3+6 = 9$. A cette date, $J_2(9,1)$ est activée et exécutée immédiatement. L'unité de capacité consommée est restituée au serveur à $t = 9+6 = 15$. Entre temps, $J_3(13,2)$ est activée mais il ne reste qu'une unité de temps de capacité. J_3 s'exécute donc en deux fois, la deuxième après le rafraîchissement dû à l'exécution de J_2. Le rafraîchissement de la capacité se fera aux dates $t = 19$ et $t = 21$ d'une unité à chaque fois.

Bien que violant, comme le serveur ajournable, la définition d'une tâche en se suspendant, le serveur sporadique permet une borne d'ordonnançabilité élevée. La méthode de rafraîchissement permet d'ailleurs la preuve du résultat suivant [93] : un jeu de tâche périodique ordonnançable contenant une tâche T_i est aussi ordonnançable si T_i est remplacée par un serveur sporadique de même durée et de même période. Ceci permet d'obtenir la borne d'ordonnançabilité suivante :

THÉORÈME 6.10 (SPRUNT, LEHOCZKY ET SHA)
Un jeu de n tâches périodiques est ordonnançable avec un serveur sporadique de capacité C_s et de

période P_s si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq \left[\left(\frac{2}{U_s + 1} \right)^{1/n} - 1 \right] \quad \text{où } U_s = C_s/P_s \tag{6.17}$$

6.3.1.5 Slack stealing

En français "slack stealing*" veut dire "vol de temps creux". Ici, il n'y a pas de tâche supplémentaire jouant le rôle de serveur. Le principe, introduit dans [115], est de déplacer (voler) les temps d'inactivité processeur (temps creux) et de les utiliser pour exécuter des tâches apériodiques souples. Les tâches périodiques étant ordonnancées avec Rate Monotonic, aux n tâches correspond donc n niveaux de priorité. On définit pour chaque niveau de priorité i une fonction $A_i(t)$ qui donne le temps cumulé d'inactivité prévu à une date t. Cette fonction est calculée hors-ligne, pour chaque niveau de priorité, sur l'ensemble de l'hyperpériode. En-ligne, on tient à jour $\mathcal{A}$, le temps cumulé que le processeur a passé à exécuter des tâches apériodiques souples, ainsi que, pour chaque niveau de priorité, I_i le temps cumulé d'inactivité processeur constaté à un niveau de priorité i (plus i est petit, plus la tâche est prioritaire). On associe au processeur un état noté j qui dépend de la tâche qu'il exécute. Cet état peut prendre $n+2$ valeurs :

- 0 : le processeur exécute une tâche apériodique souple,
- de 1 à n : le processeur exécute une tâche périodique du niveau de priorité correspondant,
- n+1 : le processeur est inactif.

Si le processeur entre dans un état j, $0 \leq j \leq n+1$ à une date t_1 et en sort à une date t_2, alors :

$$\text{Si } j = \begin{cases} 0, & \text{ajouter } t_2 - t_1 \text{ à } \mathcal{A} \\ 1, & \text{ne rien faire} \\ 2 \leq j \leq n, & \text{ajouter } t_2 - t_1 \text{ à } I_1, \ldots, I_{j-1} \end{cases} \tag{6.18}$$

Lorsqu'une tâche apériodique souple survient à une date s, on doit calculer A^* la marge disponible pour exécuter des tâches apériodiques :

$$A^*(s,t) = min_{1 \leq i \leq n}(A_i(s,t) - I_i(s)) - \mathcal{A} \tag{6.19}$$

Soit C la durée d'exécution de la tâche apériodique souple en attente. Si $A^*(s,t) \geq C$, alors C unités de temps sont allouées immédiatement à l'exécution de la tâche apériodique dans l'intervalle $[s,s+C]$, au plus haut niveau de priorité. Si $A^*(s,t) < C$, l'intervalle $[s,s+A^*(s,t)]$ est alloué pour l'exécution de la tâche apériodique souple au plus haut niveau de priorité, mais la tâche apériodique souple devra attendre que de la marge supplémentaire soit disponible pour achever son exécution. Ceci ne peut arriver que lorsqu'une tâche périodique s'est exécutée, ce qui restreint la mise à jour de $A^*(s,t)$ aux moments où une tâche apériodique souple arrive dans la file d'attente vide, ou lorsqu'une tâche périodique s'achève et qu'il y a une tâche apériodique souple en attente.

Une variante de cette technique, basée sur le même principe, mais utilisant une autre structure de données, est présentée dans [116]. Dans cette variante, l'espace mémoire nécessaire est moins important mais cela oblige à davantage de calculs en-ligne. Dans [117], les auteurs présentent le serveur à échange de priorités étendu (*Extended Priority Exchange*) qui reprend l'idée du slack

stealing mais en utilisant les principes du serveur à échange de priorités pour récupérer le temps inutilisé par les tâches apériodiques (lorsque la durée d'exécution est inférieur à C_i) grâce aux capacités des différents niveaux de priorité.

Dans [118] le slack stealing est étendu au cas des apériodiques fermes. Un test d'acceptation est présenté. Les tâches apériodiques fermes acceptées sont ordonnancées par ordre d'échéance de la plus proche à la plus tardive. Toutes les tâches apériodiques fermes sont ordonnancées au niveau de priorité le plus élevé, bien que ce ne soit pas forcément nécessaire, l'objectif étant de satisfaire les échéances et non de minimiser les temps de réponse.

La technique de slack stealing est détaillée plus longuement dans [119]. On y explique comment ordonnancer les tâches apériodiques fermes au niveau de priorité minimal, de façon à minimiser les mises à jour. Des extensions, comme la récupération du temps inutilisé ou la possibilité de mêler apériodiques souples et fermes sont également présentées.

6.3.2 Approches à priorités dynamiques

Sont ici détaillées les techniques permettant d'ordonnancer des tâches périodiques avec des tâches apériodiques souples ou fermes, en utilisant l'algorithme d'ordonnancement Earliest Deadline First pour l'ordonnancement des tâches périodiques, ainsi que les résultats d'ordonnançabilité associés. Le principal intérêt de ces techniques par rapport à celles reposant sur Rate Monotonic est la borne d'ordonnançabilité de Earliest Deadline First, qui permet d'atteindre les 100% d'utilisation processeur. Les hypothèses sur le jeu de tâches périodiques sont les mêmes que pour les approches à priorités fixes. Nous rappelons que les tâches périodiques sont notées $T_i(C_i,P_i)$, les tâches apériodiques souples $J_i(r_i,C_i)$ et les tâches apériodiques fermes $J_i(r_i,C_i,D_i)$.

6.3.2.1 Serveur à scrutation dynamique

Le serveur à scrutation dynamique* (*dynamic polling server*) [120][94] est la version à priorités dynamiques du serveur à scrutation. Ce serveur se comportant comme une tâche périodique, il suffit de dimensionner le rapport C_s/P_s pour obtenir un facteur d'utilisation de 1. C'est d'ailleurs son seul intérêt par rapport à sa version en priorités fixes car il souffre du même défaut, à savoir des temps de réponse importants. On peut adapter ce serveur au cas où les échéances des tâches périodiques sont inférieures aux périodes, mais il n'existe pas de version pour tâches apériodiques fermes.

6.3.2.2 Serveur ajournable dynamique

Le serveur ajournable dynamique* (*dynamic deferrable server*) [120] est la version à priorités dynamiques du serveur ajournable. Il partage avec la version à priorités fixes le même défaut, sa borne d'ordonnançabilité. En effet, le fait que la tâche serveur puisse se suspendre alors qu'elle est la plus prioritaire et puisse ensuite s'exécuter à tout moment l'empêche d'utiliser les résultats d'ordonnançabilité d'Earliest Deadline First. Dans [120], les auteurs donnent une condition d'ordonnançabilité pour le cas où les échéances sont quelconques.

THÉORÈME 6.11 (GHAZALIE ET BAKER)
Un jeu de n tâches périodiques est ordonnançable par Earliest Deadline First avec un serveur

ajournable dynamique de capacité C_s et de période P_s si :

$$\forall_{1,\ldots,n} k \quad \left(\sum_{i=1}^{k} \frac{C_i}{\min(D_i,P_i)}\right) + \left(1 + \frac{P_s - C_s}{D_k}\right)\frac{C_s}{P_s} \leq 1 \tag{6.20}$$

Cette condition suffisante donne une borne d'ordonnançabilité faible, même quand les échéances sont égales aux périodes. Il n'existe pas de version pour les apériodiques fermes.

6.3.2.3 Serveur à échange de priorités dynamique

Là encore, le serveur à échange de priorités dynamique* (*dynamic priority exchange*) [94] n'est que l'adaptation à priorités dynamiques du serveur à échange de priorités. Le principe reste donc le même mis à part que les capacités sont accumulées avec une certaine échéance, car les priorités dépendent des échéances.

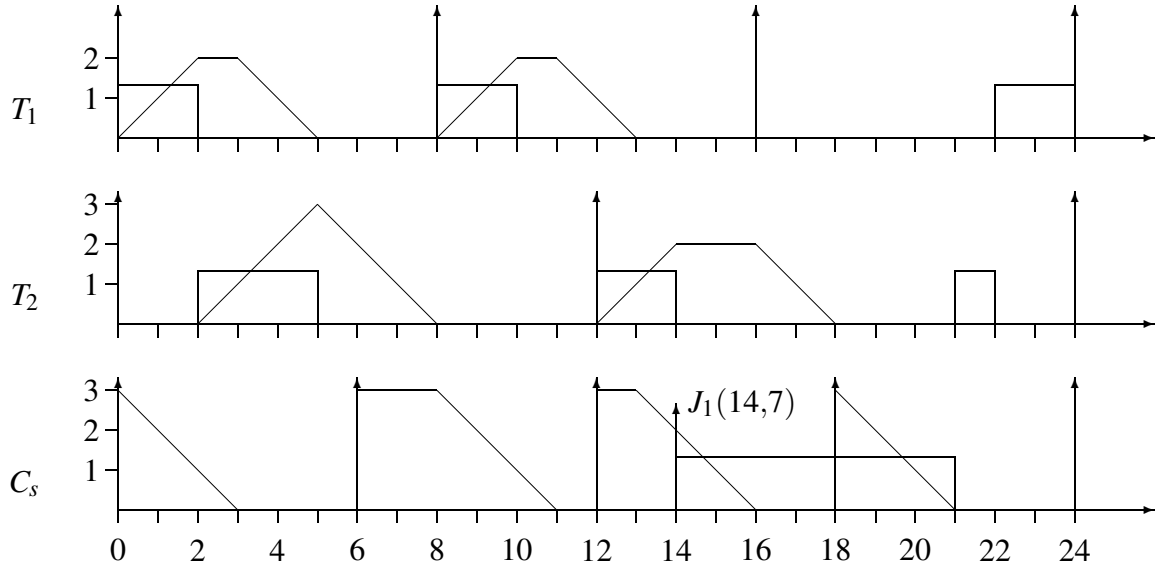

FIG. 6.8 – *Serveur à échange de priorités dynamique*

La figure 6.8 page 159 donne un exemple d'ordonnancement avec serveur à échange de priorités dynamique. Le jeu de tâches périodiques est constitué de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Le serveur a une capacité de 3 et une période de rafraîchissement de 6. En absence de tâches apériodiques souples, la capacité la plus prioritaire échange son échéance avec celle de la tâche périodique s'exécutant. Ainsi, de $t = 0$ à $t = 2$ deux unités de capacités du serveur prennent pour échéance 8 (échéance $d_{1,1}$ de T_1). Ensuite, c'est T_2 qui s'exécute entre $t = 2$ et $t = 5$. Les capacités des tâches plus prioritaires échangent successivement leurs échéances avec celle de T_2, si bien qu'à $t = 5$ les trois unités de capacités disponnibles ont pour échéance $t = 12$. A $t = 5$, il n'y aucune tâche active jusqu'à $t = 8$, la capacité est donc décrémentée progressivement. On notera que l'échéance du serveur est, à $t = 6$, égale à celle de T_2, le choix de décrémenter une des capacités

plus que l'autre est donc arbitraire. Ensuite, dans [8,10] la capacité est accumulée par T_1. Dans [10,12], en absence de tâche active, les capacités du serveur puis de T_1 sont décrémentées. Enfin sur [12,14], c'est T_2 qui en s'exécutant accumule de la capacité.

Lorsque $J_1(14,7)$ est activée, elle est servie par la capacité la plus prioritaire. Ainsi, elle utilise d'abord celle du serveur ($d_{s,3} = 18$) puis la capacité accumulée par T_2 ($d_{2,2} = 24$) avant de terminer avec celle du serveur qui a été rafraîchie. Ensuite T_2 puis T_1 peuvent s'exécuter. On remarque qu'à $t = 16$ on exécute J_1 plutôt que T_1 ou T_2 alors que les échéances de la capacité accumulée par T_2, de la tâche T_2 et de la tâche T_1 sont identiques. Cette décision ne modifie pas l'ordonnançabilité du système mais permet de minimiser le temps de réponse des tâches apériodiques souples.

L'intérêt du Dynamic Priority Exchange dans sa version à priorités dynamiques par rapport à sa version à priorités fixes est que la borne d'ordonnançabilité reste celle d'Earliest Deadline First. Ainsi, un jeu de n tâches périodiques aux échéances égales aux périodes est ordonnançable avec un serveur à échange de priorités dynamique de capacité C_s et de période P_s si et seulement si :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \frac{C_s}{P_s} \leq 1 \tag{6.21}$$

Ceci permet de définir un rapport C_s/P_s plus grand que dans le cas à priorités fixes. Dans [120], une condition d'ordonnançabilité pour les cas où les échéances sont quelconques est donnée.

De plus, le serveur à échange de priorités dynamique permet de facilement utiliser le temps processeur inutilisé par les tâches périodiques. En effet, les C_i ne sont que des durées d'exécution au pire cas, et pour la plupart des instances, cette durée n'est pas atteinte. Il suffit d'ajouter la durée non utilisée à la capacité accumulée avec cette échéance.

Il n'existe pas de version pour les apériodiques fermes.

6.3.2.4 Serveur sporadique dynamique

Le serveur sporadique dynamique* (*dynamic sporadic server*) [94] reprend les principes de sa version à priorités fixes. Il nécessite néanmoins l'ajout de règles concernant son échéance afin de garantir des temps de réponse corrects mais sans nuire à l'ordonnançabilité. Pour cela, on définit une machine à état pour le serveur. Elle comporte 3 états : en exécution (EXE), prêt (READY) et inactif (IDLE). Soit C_s la capacité du serveur et P_s sa période, à chaque transition entre les états, est attachée une action :

- IDLE $\Rightarrow$ READY : soit une tâche apériodique souple vient d'être activée et $C_s > 0$, soit C_s a été rafraîchie. Dans les deux cas, l'échéance du serveur et sa prochaine date de rafraîchissement sont définies par $d_s = t + P_s$ où t est la date courante ;
- READY $\Rightarrow$ IDLE : le serveur est la tâche la plus prioritaire mais il n'y a pas de tâche apériodique souple active ;
- READY $\Rightarrow$ EXE : le serveur est la tâche la plus prioritaire et il existe au moins une tâche apériodique souple active. La capacité disponible est consommée ;
- EXE $\Rightarrow$ READY : le serveur est préempté par une tâche plus prioritaire ;
- EXE $\Rightarrow$ IDLE : il y a deux possibilités. Soit le serveur a fini l'exécution d'une tâche apériodique souple et il n'en existe pas d'autre active, soit $C_s = 0$. Dans les deux cas, un rafraîchissement de la quantité de capacité consommée aura lieu à $t = d_s$.

Au début de l'exécution le serveur est en état READY et son échéance est égale à $t = P_s$.

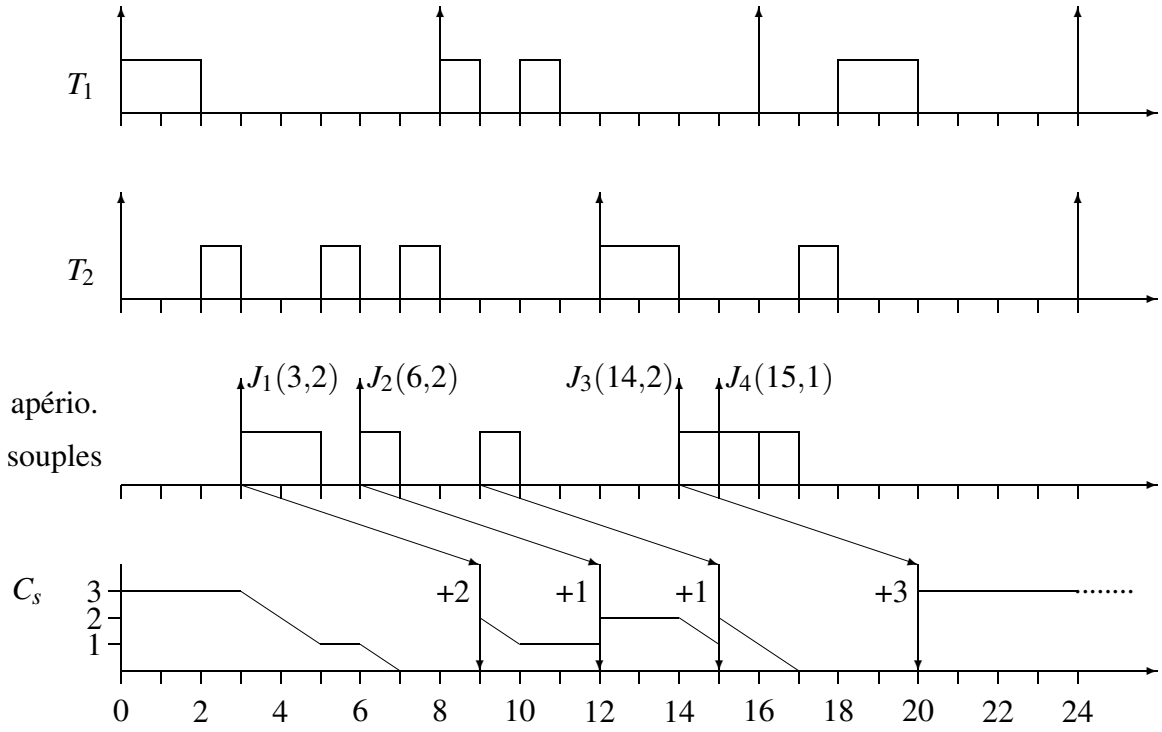

FIG. 6.9 – *Serveur sporadique dynamique*

La figure 6.9 page 161 donne un exemple d'ordonnancement avec serveur sporadique dynamique. Le jeu de tâches périodiques est constitué de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Le serveur sporadique dynamique a une capacité de 3 et une période de rafraîchissement de 6. A $t = 0$, le serveur passe de l'état READY à l'état IDLE, laissant T_1 s'exécuter. A $t = 3$, l'activation de la tâche apériodique souple $J_1(3,2)$ fait passer le serveur à l'état READY et une échéance/date de rafraîchissement est définie à $t = 3 + 6 = 9$. Le serveur, tâche la plus prioritaire, passe alors immédiatement à l'état EXE. Lorsque J_1 termine son exécution à $t = 5$, le serveur repasse à l'état IDLE, définissant ainsi le nombre d'unités de capacité à rafraîchir à $t = 2$. A $t = 6$, il ne reste qu'une unité de capacité à donner à $J_2(6,2)$, ce qui définit une échéance/date de rafraîchissement à $t = 6 + 6 = 12$ et un rafraîchissement d'une unité. Lorsque le serveur recouvre deux unités de capacité à $t = 9$, J_2 peut terminer son exécution, définissant ainsi un rafraîchissement d'une unité à l'échéance/date de rafraîchissement $t = 9 + 6 = 15$. Sur l'intervalle $[14,17]$ le serveur exécute à la suite les tâches apériodiques souples $J_3(14,2)$ et $J_4(15,1)$, sans passer par l'état IDLE, ce qui n'entraîne qu'un rafraîchissement de 3 unités de capacité à $t = 20$.

Une version nécessitant la mise à jour de davantage d'échéances mais aux temps de réponse encore meilleurs est décrite dans [120]. Est aussi présentée dans cet article une condition d'ordonnançabilité pour le cas où les échéances des tâches périodiques sont quelconques. Dans le cas où les échéances sont égales aux périodes, le résultat vu pour le serveur à priorités dynamiques s'applique aussi. Cela permet de calculer le rapport C_s/P_s facilement en fonction du facteur d'uti-

lisation processeur du jeu de tâches périodiques en faisant $C_s/P_s = 1 - \sum C_i/P_i$. C'est le cas dans notre exemple : $C_s/P_s = 1 - 2/8 - 3/12 = 0.5$. Pour comparaison, dans le cas du serveur sporadique à priorités fixes, la borne d'ordonnançabilité présentée au 6.3.1.4 page 157 ne nous permet de fixer C_s/P_s qu'à 0.28.

Il a néanmoins un défaut par rapport à sa version à priorités fixes. En effet, le choix de la période du serveur, une fois le rapport C_s/P_s défini, n'est pas facile. Choisie trop grande, cette période entraîne des échéances tardives, ce qui nuit à la priorité du serveur, et donc aux temps de réponse des tâches apériodiques souples. Choisie trop petite, elle entraîne beaucoup de rafraîchissement et donc de calculs supplémentaires en-ligne.

Il n'existe pas de version de ce serveur pour les apériodiques fermes.

6.3.2.5 Serveur à utilisation totale

Le serveur à utilisation totale* (*total bandwidth server*) [94] tend à résoudre le problème de priorité du serveur sporadique dynamique en assignant des échéances aux tâches apériodiques souples. Cet assignement doit néanmoins garantir que le facteur d'utilisation processeur des tâches apériodiques souples ne dépasse pas une valeur spécifiée U_s. Ce taux d'utilisation est calculé en utilisant le résultat d'ordonnançabilité d'Earliest Deadline First, $U_s = 1 - \sum C_i/P_i$.

Quand la $k^{ième}$ tâche apériodique souple est activée à $t = r_k$, elle se voit affecter une échéance d_k telle que :

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{6.22}$$

où $d_0 = 0$. Les tâches apériodiques souples sont ensuite placées dans la même file d'attente que les tâches périodiques et exécutées par ordre croissant d'échéance.

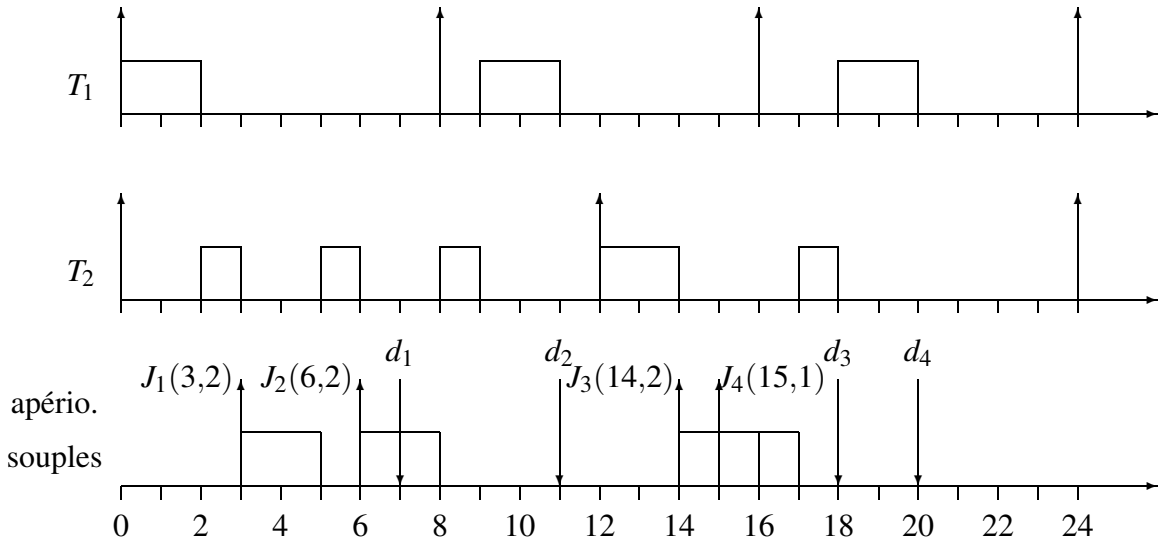

FIG. 6.10 – *Serveur à utilisation totale*

La figure 6.10 page 162 donne un exemple d'ordonnancement avec serveur à utilisation totale. Le jeu de tâches périodiques est constitué de deux tâches $T_1(2,8)$ et $T_2(3,12)$. Ce jeu de tâche est identique à celui de l'exemple concernant le serveur sporadique dynamique. On choisit donc $U_s = 1 - \sum C_i/P_i = 0.5$. Les dates d'activation et les durées d'exécution des tâches apériodiques souples sont également identiques. Lorsque $J_1(3,2)$ est activée, elle reçoit comme échéance $d_1 = max(3,0) + \frac{2}{0.5} = 7$ puis est exécutée immédiatement car il s'agit de l'échéance la plus proche. Les autres tâches apériodiques souples reçoivent leurs échéances de la même façon : $d_2 = max(6,7) + \frac{2}{0.5} = 11$, $d_3 = max(14,11) + \frac{2}{0.5} = 18$ et $d_4 = max(15,18) + \frac{1}{0.5} = 12$.

En comparant cet exemple à la figure 6.9 page 161 illustrant les performances du serveur sporadique dynamique pour le même scénario, on s'aperçoit que le temps de réponse de la tâche J_2 est amélioré. En s'intéressant aux échéances assignées dans les deux cas, on s'aperçoit que les échéances assignées aux quatre tâches apériodiques souples par le serveur à utilisation totale (7,11,18,20) sont toutes inférieures ou égales à celles assignées par le serveur sporadique dynamique (9,12,20,20). Cela permet un traitement plus prioritaire aux tâches apériodiques souples.

Au niveau implémentation cette technique est simple. Elle réunit les deux types de tâche dans une même file d'attente et le calcul de l'échéance est en temps constant.

Dans [90], il est montré comment le serveur à utilisation totale peut être utilisé pour l'exécution de tâches apériodiques fermes. L'idée est de comparer l'échéance assignée par le serveur à utilisation totale à celle de la tâche apériodique ferme. Si elle est inférieure ou égale, la tâche est acceptée, sinon elle est rejetée. Dans la version normale du serveur à utilisation totale, les tâches apériodiques ne peuvent pas se préempter entre elles. Afin d'améliorer le taux de tâches apériodiques fermes acceptées, la préemption est ici autorisée. La nouvelle tâche apériodique ferme est insérée dans une queue comprenant les tâches apériodiques fermes préalablement acceptées. Les tâches y sont ordonnées par échéances absolues croissantes (celles d'origine et non pas celles assignées par le serveur). Si la première tâche est la nouvelle et que l'une des tâches apériodiques fermes était en cours d'exécution, elle est préemptée et la fin de son exécution est considérée comme une nouvelle tâche. On assigne ensuite à chaque tâche son échéance et la nouvelle tâche est acceptée si pour toute tâche, l'échéance assignée est inférieure ou égale à celle d'origine.

6.3.2.6 Serveur Earliest Deadline Late

Le principe du serveur Earliest Deadline Late* (*Earliest Deadline Late server*) [94] repose sur un résultat de Chetto et Chetto [121]. Dans cet article on compare les temps d'inactivité processeur (durée, répartition) produits par l'algorithme d'ordonnancement Earliest Deadline First tel qu'on le connaît et sa version "au plus tard" nommée EDL pour Earliest Deadline Late. Dans cet algorithme d'ordonnancement, l'ordre d'exécution des tâches reste inchangé mais chaque tâche est exécutée le plus tard possible. La figure 6.11 page 164 représente un exemple d'ordonnancement avec EDL pour deux tâches $T_1(3,6)$ et $T_2(3,8)$.

Les résultats d'ordonnançabilité de Earliest Deadline First s'appliquent aussi à Earliest Deadline Late. Si l'implémentation d'un tel ordonnanceur pose problème, cet algorithme a une propriété intéressante. Soit $\Omega_E^X(t_1,t_2)$ la durée cumulée sur l'intervalle de temps $[t_1,t_2]$ des temps d'inactivité processeur obtenu par l'ordonnancement d'un jeu de tâches quelconques (périodiques et/ou apériodiques) E par l'algorithme d'ordonnancement X. Dans [121], les auteurs prouvent le résultat

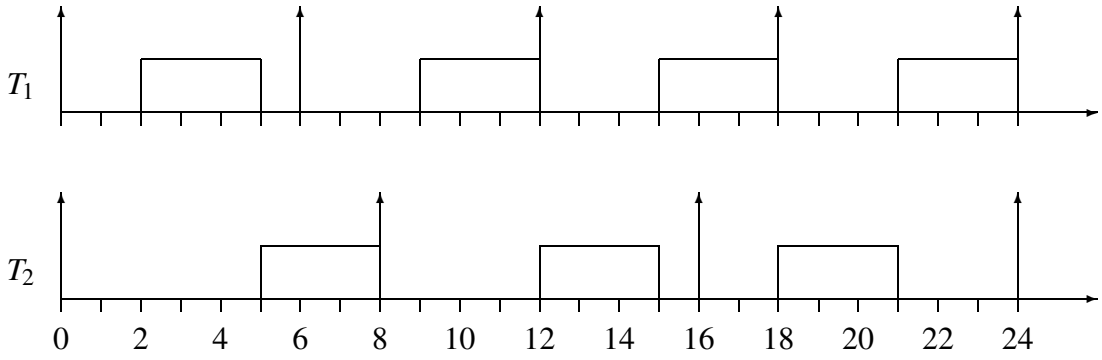

FIG. 6.11 – *Earliest Deadline Late*

suivant.

THÉORÈME 6.12 (CHETTO & CHETTO)
Soit E un jeu de tâches quelconques et A un algorithme d'ordonnancement préemptif. A tout instant t,

$$\Omega_E^{EDL}(0,t) \geq \Omega_E^A(0,t) \tag{6.23}$$

Dans ce même article, les auteurs utilisent ce résultat pour l'ordonnancement mixte de tâches périodiques et apériodiques fermes. Lorsqu'une tâche apériodique ferme est activée, on calcule $\Omega_{EDL}^X(r,d)$ où r et d sont respectivement la date d'activation et l'échéance de la tâche apériodique ferme. Si cette valeur est inférieure à la durée d'exécution de la tâche, elle est rejetée, sinon elle est acceptée. Les tâches sont ensuite ordonnancées avec Earliest Deadline First. Cette technique nécessite pourtant une contrainte forte sur les tâches apériodiques fermes, à savoir, qu'une tâche apériodique ferme ne peut être activée qu'après exécution de la tâche apériodique ferme acceptée précédente (il n'y a jamais plus d'une tâche apériodique ferme en attente).

Les mêmes auteurs proposent dans [122] une technique qui lève cette contrainte mais consiste à vérifier, à l'activation d'une tâche apériodique ferme s'il existe un ordonnancement correct de toutes les tâches sur $[r,d_{max}+Hp]$ où r est la date à laquelle la tâche apériodique ferme survient, d_{max} est la date d'échéance la plus tardive parmi les tâches apériodiques fermes acceptées et celle en cours d'acceptation, enfin Hp est l'hyperpériode de toutes les tâches apériodiques fermes acceptées. Ce test est coûteux car en $O(N)$ où N est le nombre d'instances de tâches périodiques ou apériodiques fermes devant s'exécuter avant $D+H_p$. Une approche de complexité semblable est décrite dans [123]. Enfin, dans [3], ils proposent un test dont la complexité est moindre. Cette technique peut être vue comme une version à priorités dynamiques du Slack Stealing présenté précédemment.

Le problème des tâches apériodiques souples est abordé dans [94] qui définit le serveur Earliest Deadline Late. L'idée est toujours la même, à savoir ordonnancer les tâches périodiques avec Earliest Deadline First et, à l'activation d'une tâche apériodique souple, calculer les temps d'inactivité que produirait Earliest Deadline Late afin d'exécuter les tâches apériodiques souples en attente. La méthode pour calculer les temps d'inactivité est celle présentée dans [121]. Cette méthode souffre d'une forte complexité mais est néanmoins optimale en ce qui concerne le temps de réponse.

Ainsi, bien que difficilement utilisable car trop coûteux, le serveur Earliest Deadline Late permet de définir une borne inférieure du temps de réponse et fournir une base pour des algorithmes proches de l'optimalité.

6.3.2.7 Serveur à échange de priorités dynamiques amélioré

L'idée du serveur à échange de priorités dynamiques amélioré* (*improved priority exchange*) est d'utiliser les temps d'inactivité processeur obtenus avec Earliest Deadline Late pour définir le rafraîchissement du serveur.

Hors-ligne, on calcule les périodes d'inactivité processeur obtenues par un ordonnancement avec Earliest Deadline Late sur l'hyperpériode. Leurs dates et durées sont stockées dans un vecteur de couple (e_i,δ_i) où e_i est la date à laquelle débute la $i^{ème}$ période d'inactivité et δ_i sa durée. En-ligne, la capacité du serveur est initialement nulle mais pour chaque couple du vecteur, un rafraîchissement de la capacité du serveur de e_i unités a lieu à $t = \delta_i + kHp$ avec $K \in \mathbb{N}$. De plus, le serveur est toujours la tâche la plus prioritaire, quelles que soient les échéances des autres tâches. Les échanges de priorités et l'accumulation de capacité aux différents niveaux de priorité se font selon les principes énoncés pour le serveur à échange de priorités dynamiques.

L'implémentation de ce serveur n'est pas plus compliquée que celle de serveur à échange de priorités dynamiques. Par contre cette méthode nécessite le stockage du vecteur des périodes d'inactivité dont la taille est directement liée à celle de l'hyperpériode. L'espace mémoire nécessaire peut donc être important.

6.3.2.8 Versions optimales et améliorées du serveur à utilisation totale

Dans [124], les auteurs présentent une version optimale du serveur à utilisation totale* (*total bandwidth server star*). Cette version repose sur le constat que les échéances assignées par le serveur à utilisation totale non amélioré peuvent être parfois avancées sans que cela ne remette en cause l'ordonnançabilité. Par contre, avancer une échéance peut permettre à une tâche apériodique souple d'être plus prioritaire et de terminer son exécution plus tôt, réduisant du même coup son temps de réponse. La technique consiste à calculer récursivement chaque nouvelle échéance en remplaçant à l'itération k, l'échéance d par la date de fin d'exécution de la tâche si elle était ordonnancée avec l'échéance obtenue à l'itération $k-1$. Le processus s'arrête lorsque l'échéance reste inchangée. Cette technique s'appuie sur ce premier résultat :

LEMME 1
Soit σ un ordonnancement correct d'un jeu de tâche E où une échéance d_k est allouée à une tâche apériodique souple J_k, et soit f_k la date de fin d'exécution de J_k dans σ. Si on remplace d_k par $d'_k = f_k$, alors le nouvel ordonnancement σ' produit par Earliest Deadline First est lui aussi correct.

Étant donné que la date de fin d'exécution exacte d'une tâche serait coûteux à calculer, les auteurs de [124] donnent une formule permettant d'en obtenir une borne supérieure. Ils prouvent aussi que l'échéance obtenue à la fin de la récurrence en utilisant cette borne supérieure est bien la date de fin d'exécution réelle. La complexité de ce calcul est pseudo-polynomiale car en $O(nN)$ où n est le nombre de tâches périodiques et N le nombre d'itérations nécessaires pour obtenir

l'échéance égale à la date de fin d'exécution. Cette méthode est optimale comme le montre le théorème suivant.

THÉORÈME 6.13 (BUTTAZZO & SENSINI)
Soit σ un ordonnancement correct d'un jeu de tâches E par Earliest Deadline First et f_k la date de fin d'exécution d'une tâche apériodique souple J_k ordonnancée dans σ avec une échéance d_k. Si $f_k = d_k$, alors $f_k = f_k^*$ où f_k^* est la plus petite date de fin d'exécution atteignable pour cette tâche, quel que soit l'ordonnancement.

Cette version optimale est souvent notée TBS*. Là encore, la méthode n'est pas raisonnablement implémentable car de trop grande complexité, mais en fixant le nombre d'itérations maximal, on obtient une complexité polynomiale dont les gains sur les temps de réponse, même pour des valeurs basses du nombre d'itérations, sont importants. Cette version du serveur, appelée version améliorée du serveur à utilisation totale*, est souvent notée TBS(k), où k est le nombre d'itérations.

6.3.3 Approches indépendantes de l'algorithme d'ordonnancement

6.3.3.1 Slot shifting

La technique de slot shifting* [2][125] repose sur un ordonnancement hors-ligne des tâches périodiques. Sur l'ordonnancement produit, les échéances sont modifiées de manière à ce que chaque tâche ait une échéance inférieure ou égale à la tâche qui la suit. L'ensemble des échéances définit alors des intervalles appelés intervalles d'exécution. Toujours hors-ligne, on calcule pour chaque intervalle d'exécution le temps disponible pour l'exécution de tâches apériodiques appelé spare capacity* et noté sc.

En-ligne, ces spare capacities permettent d'exécuter des tâches apériodiques souples et fermes. En effet, une tâche apériodique ferme peut être acceptée moyennant un test d'acceptation en $O(I)$ où I est le nombre d'intervalles d'exécution séparant l'intervalle d'exécution courant de l'intervalle d'exécution auquel appartient l'échéance considérée. Si la tâche est acceptée, les spare capacities sont mises à jour en utilisant le résultat de Chetto et Chetto sur Earliest Deadline Late [121], c'est-à-dire en prévoyant son exécution au plus tard. Les tâches apériodiques fermes acceptées sont ensuite ordonnées par ordre croissant d'échéance.

En-ligne, un ordonnanceur utilise Earliest Deadline First et la spare capacity de l'intervalle courant pour décider de la tâche à exécuter :

- si $sc > 0$ et qu'il y des tâches apériodiques souples en attente, la première de la file est exécutée ,
- sinon, Earliest Deadline First est utilisé pour choisir entre la tâche périodique active (si elle existe) et la première tâche apériodique ferme de la file.

On peut voir cette méthode comme une version du slack stealing utilisant l'approche hors-ligne de manière à réduire les calculs en-ligne.

Dans [126], les auteurs proposent une méthode pour, une fois l'ordonnancement hors-ligne des tâches périodiques effectué, accepter ou rejeter un ensemble de tâches sporadiques en utilisant le slot shifting. D'abord, pour chaque intervalle d'exécution, on calcule un temps critique t_c. Ensuite,

on teste l'acceptation de chaque tâche sporadique en prenant successivement pour date d'activation les temps critiques calculés précédemment. Bien sûr les spare capacities sont mises à jour en fonction des tâches sporadiques acceptées. Il est démontré que si ces tâches sont acceptées aux instants critiques, elles le seront forcément, quelles que soient leurs dates d'activation.

Dans [127], les auteurs proposent un test d'acceptation ne nécessitant pas les mises à jour des spare capacities après qu'une tâche apériodique ferme ait été acceptée. Cela diminue le surcoût en-ligne du slot shifting. Ce test peut être utilisé conjointement avec la méthode d'ordonnancement des tâches sporadiques citée précédemment comme cela est décrit dans [128].

6.3.3.2 Feedback scheduling

Le feedback scheduling* est une méthode permettant en-ligne de modifier l'ordonnancement en utilisant la théorie des lois de commande (*feedback control*). L'idée principale est de maintenir un certain degré de performance (facteur d'utilisation, taux d'acceptation des tâches apériodiques fermes ...) en agissant sur différents paramètres.

Les travaux sur ce type d'ordonnancement concernent principalement des applications où certaines grandeurs varient. Si, par exemple, la durée d'exécution des tâches varie avec les données à traiter, on peut modifier les périodes des tâches pour maintenir les performances. Dans [129][130], les auteurs appliquent ce principe pour maintenir les performances de lois de commande implémentées sous forme de tâches.

Le feedback scheduling a ainsi intéressé les développeurs de serveurs web. En effet ces serveurs sont appelés à répondre à un flux de demandes pouvant varier fortement. Dans le cas de vidéo à la demande, par exemple, le temps de réponse est important et une certaine qualité de service doit être maintenue. Dans [131], une méthode est présentée pour accepter ou rejeter des tâches apériodiques fermes et cela grâce à une loi de commande tentant de maintenir le serveur à un certain niveau de demandes acceptées (sous la forme du pourcentage, noté *MR* de requêtes satisfaites avant leurs échéances par rapport à l'ensemble des requêtes). Il est important de noter qu'ici une tâche acceptée est exécutée mais pas forcément avant son échéance. Dans [132], une extension distribuée est proposée. Un contrôleur modifie les *MR* des différents serveurs en fonction de la proportion des tâches exécutées avant leur échéance par rapport à l'ensemble des tâches acceptées. On a donc un contrôleur distribué qui paramètre les contrôleurs locaux de chaque serveur.

Bien que proche de nos préoccupations, il n'est néanmoins pas possible d'utiliser le feedback scheduling pour mêler temps réel strict et temps réel souple, une forte variation provocant forcément la violation de contraintes.

6.3.4 Cas distribué

Les tâches périodiques sont maintenant exécutées sur une architecture distribuée constituée de plusieurs processeurs reliés par des médias de communication. La distribution des tâches périodiques a été effectuée hors-ligne. Il s'agit alors d'essayer d'exécuter en plus des tâches apériodiques.

La plupart du temps le problème est ramené à un problème d'ordonnancement monoprocesseur. Ainsi, on considère habituellement qu'une tâche apériodique survient sur un processeur et ne peut

être exécutée que par ce processeur [122][125]. Sous ces conditions, l'ensemble des méthodes décrites précédemment peuvent être appliquées, si l'algorithme d'ordonnancement le permet. Si on traduit l'ordonnancement hors-ligne en un jeu de tâches à priorités fixes, ces priorités traduisent l'ordonnancement calculé hors-ligne et ne sont pas assignées par Rate Monotonic. Les méthodes associées ne sont donc ici d'aucune utilité. Par contre, nous avons vu qu'un ordonnancement hors-ligne pouvait être transformé en un jeu de tâches à ordonnancer avec Earliest Deadline First [112], ce qui permet d'utiliser les approches à priorités dynamiques (voir 6.3.2 page 158).

La migration des tâches apériodiques d'un processeur à un autre est abordée dans [87]. Ici le processeur sur lequel la tâche apériodique est activée peut la rejeter et décider de l'envoyer à un autre processeur. Pour choisir le processeur auquel faire suivre la tâche, un algorithme de requête et d'élection est proposé. Il nécessite de nombreuses communications, sur des média déjà utilisés pour les communications entre tâches périodiques. Si le taux d'utilisation de ces média par les communications entre tâches périodiques est significatif, le gain de la migration (plus de tâches acceptées, temps de réponse raccourcis) est fortement atténué à cause de la congestion qu'elle entraîne sur les média de communication.

Pour éviter cette surcharge de communication, l'autre solution est d'avoir une connaissance globale de l'état du système : tâches périodiques allouées à chaque processeur, disponibilité de chaque processeur, tâches apériodiques en attente ou en cours d'exécution sur les différents processeurs. Une possibilité est de donner ce rôle à l'un des processeurs. Ce processeur est alors le seul à pouvoir choisir sur quel processeur doit être exécutée chaque nouvelle tâche apériodique. La contre partie est que la charge de calcul de ce processeur peut varier fortement en fonction du flux de tâches apériodiques et il ne peut donc pas exécuter d'autres tâches sans risquer de manquer leurs échéances. Cette solution revient donc à rajouter un processeur supplémentaire à l'architecture sur lequel n'est ordonnancée aucune tâche et peut être comparé au fait de rajouter un processeur sur lequel s'exécuterait toutes les tâches apériodiques.

6.3.5 Optimalité et performances

6.3.5.1 Optimalité

Concernant les tâches apériodiques souples, l'optimalité d'un algorithme se juge à sa capacité à minimiser les temps de réponse de chaque tâche apériodique souple, ou le temps de réponse moyen.

Dans les approches serveur à priorités fixes, les bornes d'ordonnançabilité ne permettent pas, quel que soit le jeu de tâches périodiques, d'utiliser la totalité du temps disponible pour exécuter des tâches apériodiques. Le seul algorithme pouvant prétendre à l'optimalité est donc le slack stealing. Dans [115], les auteurs prouvent son optimalité :

THÉORÈME 6.14 (RAMOS-THUEL ET LEHOCZKY)
Soit un ensemble de tâches périodiques ordonnancées par un algorithme à priorités fixes et un flux de tâches apériodiques souples ordonnées en FIFO. L'algorithme de Slack Stealing minimise le temps de réponse de chaque tâche apériodique souple par rapport à tout autre algorithme d'ordonnancement qui respecte les échéances.

Mais, en absence de plus de restrictions, ce résultat est faux. L'optimalité n'est que locale, le temps de réponse ne pouvant être minimisé que pour la première tâche apériodique souple de la file d'attente. Ceci fût prouvé dans [133] et s'illustre par l'exemple suivant. Le jeu de tâches périodiques est composé de trois tâches $T_1(1,3)$, $T_1(1,4)$ et $T_1(1,6)$ ordonnancées avec Rate Monotonic. Considérons deux tâches apériodiques souples $J_1(2,1)$ et $J_2(3,1)$. Lorsque la première est activée, il y a du temps disponible et elle peut donc être exécutée sur l'intervalle $[2,3]$. Mais dans ce cas, lorsque la deuxième est activée, il n'y a pas de temps disponible avant $t = 6$, ce qui donne l'ordonnancement illustré par la figure 6.12 page 169. Si on considère ensuite l'ordonnancement illustré par la figure 6.13 page 170, on s'aperçoit qu'en différant l'exécution de J_1, le temps de réponse de J_2 peut être réduit. Il est donc impossible de minimiser à la fois le temps de réponse de J_1 et celui de J_2, ce qui conduit au théorème suivant :

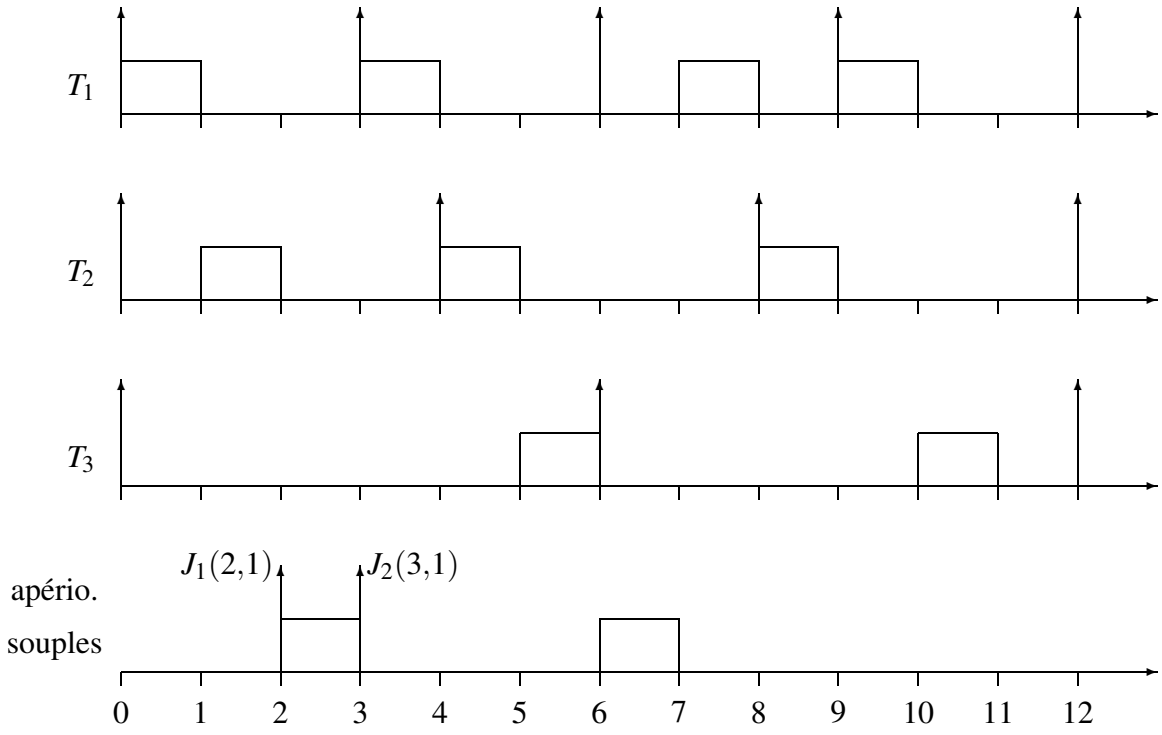

FIG. 6.12 – *Exemple A pour la preuve de non-optimalité*

THÉORÈME 6.15 (TIA, LIU ET SHANKAR)
Soit un ensemble quelconque de tâches périodiques ordonnancées par un algorithme à priorités fixes et des tâches apériodiques souples ordonnées par une politique de file d'attente donnée. Il n'existe aucun algorithme qui minimise le temps de réponse de chaque tâche apériodique souple.

Ce résultat s'applique à tous les algorithmes même ceux qui sont clairvoyants, c'est-à-dire ont connaissance des activations de tâches apériodiques souples à venir. Si on se restreint aux algorithmes non clairvoyants, on ne peut atteindre l'optimalité en ce qui concerne le temps de réponse

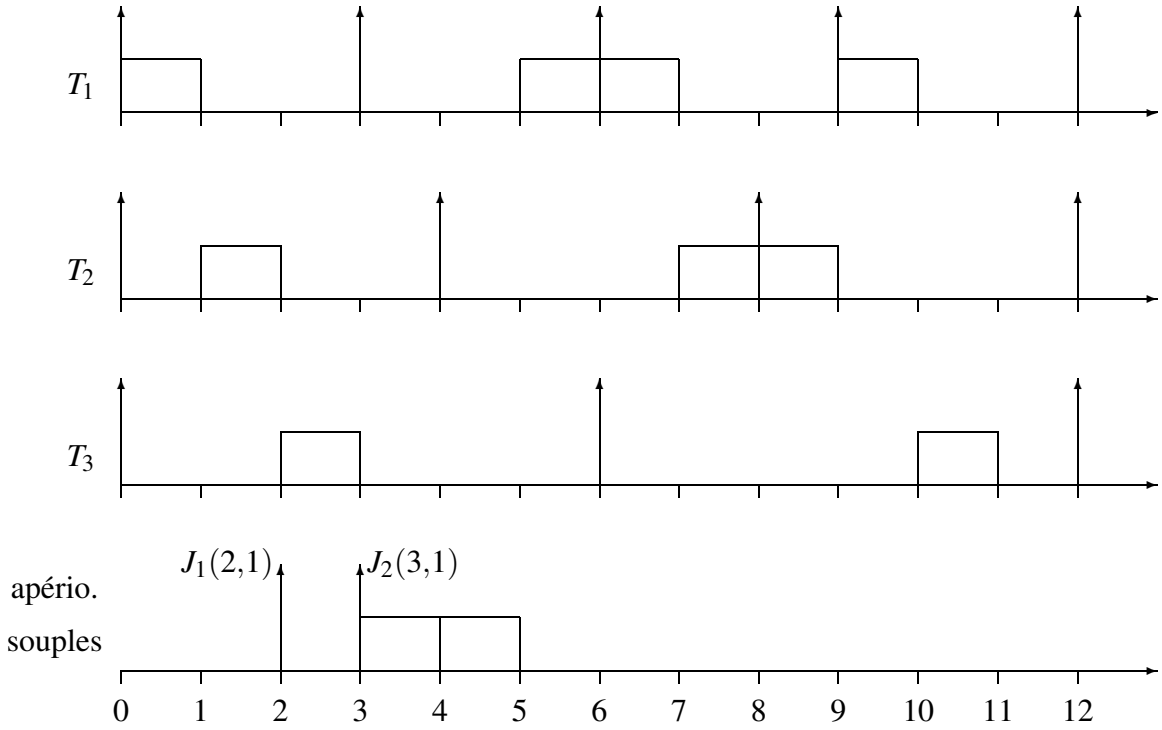

FIG. 6.13 – *Exemple B pour la preuve de non-optimalité*

moyen :

THÉORÈME 6.16 (TIA, LIU ET SHANKAR)
Soit un ensemble quelconque de tâches périodiques ordonnancées par un algorithme à priorités fixes et des tâches apériodiques souples ordonnées par une politique de file d'attente donnée. Il n'existe aucun algorithme non clairvoyant qui minimise le temps de réponse moyen des tâches apériodiques souples.

Pour démontrer ce théorème, il suffit de considérer le même exemple. Si J_1 survient seul, la décision doit être différente du cas où J_2 survient également, ce qui, pour un algorithme non clairvoyant, n'est pas possible.

Les auteurs concluent en donnant une définition de l'optimalité locale, qui elle est atteignable avec les algorithmes à priorités fixes. Un algorithme atteint l'optimalité locale s'il minimise le temps de réponse de la première opération de la file d'attente. Le slack stealing est un algorithme qui atteint cette optimalité locale.

Dans les approches de type serveur à priorités fixes, c'est l'impact des tâches apériodiques souples sur les tâches à faibles priorités qui empêche d'atteindre l'optimalité. Sur l'exemple précédent, le fait que l'on n'exécute pas T_3 à $t = 2$ repousse son exécution à $t = 5$, après les exécutions des deuxièmes instances des tâches T_1 et T_2. Pourtant de part son échéance, T_3 est plus urgente, ce qui empêche toute tâche apériodique de s'exécuter avant elle. Ce problème n'existe pas avec les algorithmes à priorités dynamiques. La figure 6.14 page 171 montre le même jeu de tâches mais

ordonnancé cette fois avec Earliest Deadline First et la version améliorée du serveur à utilisation totale, qui est optimale. L'autre serveur optimal est Earliest Deadline Late. Dans [2], l'équivalence entre Earliest Deadline Late et slot shifting est démontrée et par la même, l'optimalité de cette méthode.

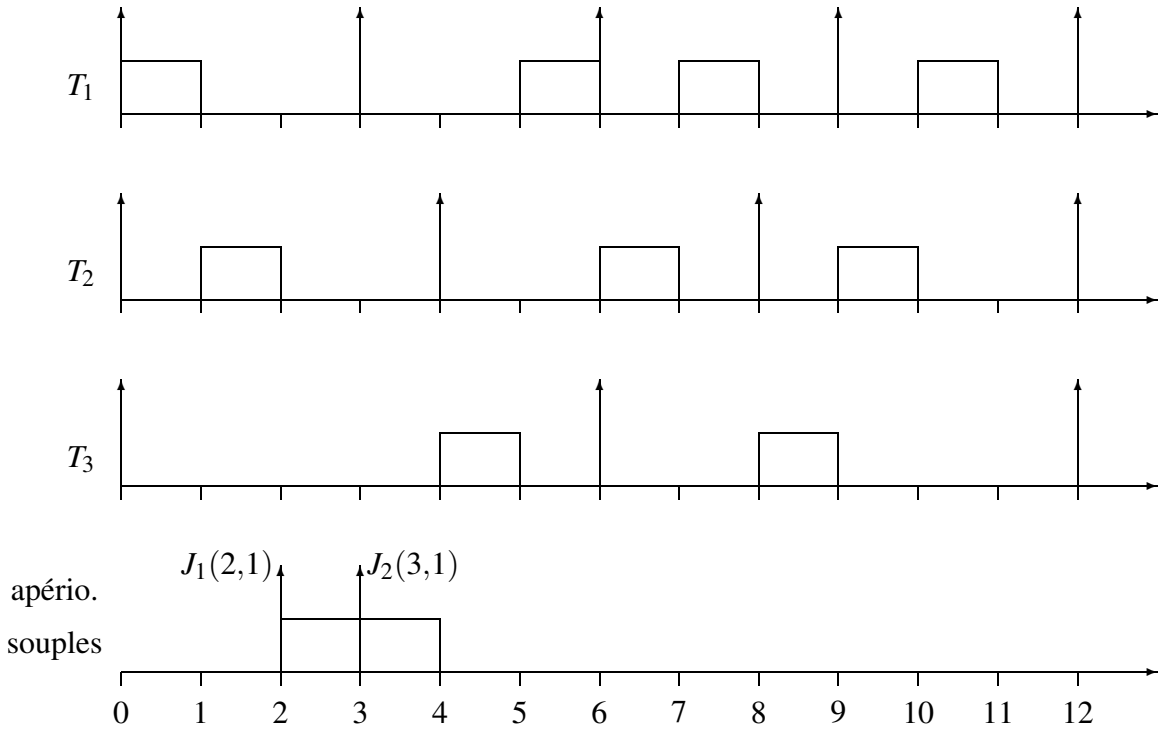

FIG. 6.14 – *Exemple d'optimalité de la version améliorée du serveur à utilisation totale*

Pour ce qui est des tâches apériodiques fermes, il est démontré dans [118] qu'il n'y a pas d'algorithme d'ordonnancement de tâches apériodiques fermes qui soit optimal, au sens où quel que soit un jeu de tâches apériodiques fermes, si un algorithme permet de respecter l'échéance d'une des tâches, cette échéance serait respectée avec l'algorithme optimal.

6.3.5.2 Performances des approches à priorités fixes

Les performances des différentes méthodes peuvent être jugées sur quatre critères [5][95] : les temps de réponse des tâches apériodiques souples, la borne d'ordonnançabilité, la complexité d'implémentation et enfin le surcoût en-ligne.

Les temps de réponse des tâches apériodiques souples dépendent non seulement des serveurs mais aussi des caractéristiques des tâches apériodiques souples : durée moyenne par rapport à la capacité du serveur, charge processeur générée, etc. Néanmoins, il apparaît que derrière le slack stealing, c'est le serveur ajournable, le serveur à échange de priorités puis le serveur sporadique qui donnent les meilleurs temps de réponse. L'écart de performances entre les deux derniers croît avec la charge processeur engendrée par les tâches apériodiques souples, c'est-à-dire avec le nombre

et l'inverse de la durée moyenne séparant les activations des tâches apériodiques. Ces temps de réponse restent néanmoins très inférieurs à ceux obtenus avec le serveur à scrutation et l'ordonnancement en tâche de fond. Ces résultats sont illustrés par les figures 6.3.5.2 page 172 et 6.3.5.2 page 173 qui comparent les temps de réponse moyens obtenus par l'ordonnancement en tâche de fond (BS), le serveur à scrutation (Polling), le serveur ajournable (DS), le serveur sporadique (SS) et le serveur à échange de priorités (PE).

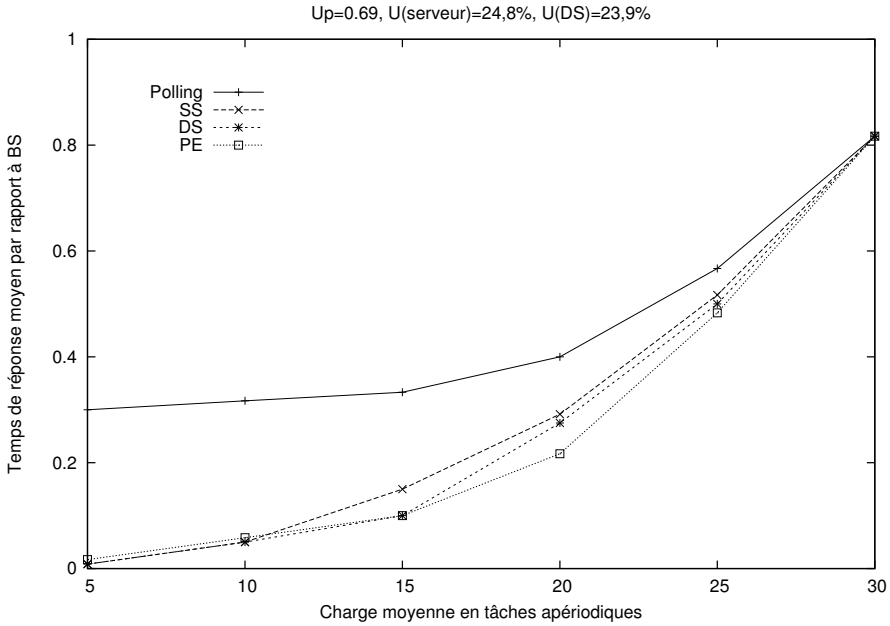

FIG. 6.15 – *Temps de réponse moyens relativement à BS en fonction de la charge due aux tâches apériodiques*

Les bornes d'ordonnançabilité sont fréquemment utilisées pour dimensionner les serveurs, c'est-à-dire définir la capacité et la période de ceux-ci. Une borne d'ordonnançabilité élevée permettra donc de traiter davantage de tâches apériodiques souples. Le slack stealing et l'ordonnancement en tâche de fond permettent tous les deux d'atteindre des facteurs d'utilisation de 100%. Parmi les serveurs, c'est le serveur sporadique qui a la borne la plus élevée, viennent ensuite le serveur à échange de priorités, le serveur à scrutation et le serveur ajournable.

Au niveau de la complexité d'implémentation, l'ordonnancement en tâche de fond, le serveur à scrutation et le serveur ajournable sont relativement facile à implémenter, quel que soit le système d'exploitation temps réel. Le serveur sporadique et le serveur à échange de priorités sont plus difficiles à implémenter (méthode de rafraîchissement pour le premier, multiples capacités et échanges pour le second), le premier étant tout de même plus simple que le second. La solution la plus complexe à implémenter est le slack stealing à cause du calcul de la fonction $A^*(s,t)$. On peut noter que du point de vue de l'espace mémoire nécessaire, le slack stealing est le plus gourmand mais que le serveur à échange de priorités, parce qu'il double le nombre de tâches (capacités à chaque niveau de priorité), a aussi un impact important.

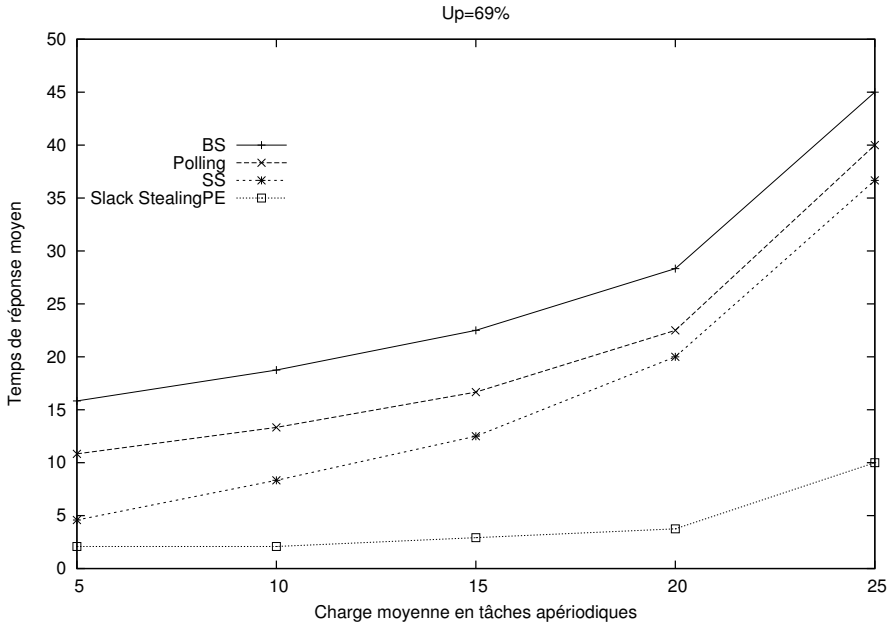

FIG. 6.16 – *Temps de réponse moyens en fonction de la charge due aux tâches apériodiques*

Enfin, le surcoût en-ligne de ces méthodes correspond en grande partie à leur complexité d'implémentation. Il est négligeable pour l'ordonnancement en tâche de fond, le serveur à scrutation et le serveur ajournable, tolérable pour le serveur sporadique et le serveur à échange de priorités, et important pour le slack stealing.

La conclusion usuelle de ces observations est que si on se restreint aux méthodes utilisant des serveurs, la complexité du slack stealing l'éliminant d'office, le serveur sporadique est le plus performant car sa borne d'ordonnançabilité est plus élevée que le serveur ajournable, sa complexité d'implémentation moindre que le serveur à échange de priorités et les temps de réponse obtenus sont semblables.

Il faut néanmoins relativiser ces conclusions. En effet, la borne d'ordonnançabilité n'est pas forcément un bon outil de comparaison. Dans [134], les auteurs ont comparé le serveur ajournable et le serveur sporadique. On reproche d'habitude au premier sa borne d'ordonnançabilité, mais il s'avère que pour un jeu de tâches périodiques donné, le facteur d'utilisation maximal du serveur ajournable est fréquemment le double que celui indiqué par la borne d'ordonnançabilité usuelle. De plus, il est rare qu'il diffère beaucoup de celui atteignable avec un serveur sporadique. Une méthode de calcul des pire temps de réponse, plus complète que celle présentée dans [120] est donnée.

En ce qui concerne les temps de réponse des tâches apériodiques souples, il est indiqué qu'ils sont d'autant plus petits que la capacité du serveur est grande. On a donc tout intérêt à augmenter la période du serveur. Le serveur doit avoir la plus grande priorité mais pas forcément la plus petite période.

De plus, cet article révèle un avantage du serveur ajournable sur le serveur sporadique. Consi-

dérons le cas d'un serveur ajournable dont la capacité est entière C_s unité de temps avant son rafraîchissement. Si une tâche apériodique souple est activée à cette même date, avec une durée C telle que $C_s < C \leq 2C_s$, elle va pouvoir s'exécuter entièrement en une seule fois. Cette situation, appelée double blocage et déjà mentionnée dans [120], ne survient qu'avec le serveur ajournable et est en grande partie responsable de la faible borne d'ordonnançabilité, mais permet de meilleur temps de réponse pour des tâches apériodiques souples ayant une grande durée d'exécution.

La conclusion de cet article est que l'on peut atteindre les même performances (facteur d'utilisation et temps de réponse) avec une serveur ajournable qu'avec un serveur sporadique. Il suffit de choisir plus précisément ses caractéristiques, d'autant plus que sa complexité d'implémentation et son surcoût en-ligne sont moindres.

Enfin, il est intéressant d'observer que malgré le coût mémoire important de son implémentation, le slack stealing permet des temps de réponse inatteignables par les autres approches. Les différences observées sont maximales dans le cas où le facteur d'utilisation du processeur par les tâches périodiques est important [115]. Au contraire, dans le cas où cette charge est faible et que le flux de tâches apériodiques souples est important, les temps de réponse diffèrent peu de ceux obtenus avec un serveur sporadique par exemple. De plus, c'est dans ce cas que la complexité de cet algorithme entraîne un surcoût important.

6.3.5.3 Performances des approches à priorités dynamiques

La comparaison est beaucoup plus simple pour les approches à priorités dynamiques bien qu'elles soient plus nombreuses. En effet, première différence, la borne d'ordonnançabilité est de 100% pour toutes les méthodes hormis le serveur ajournable dynamique, uniquement présenté ici pour des raisons d'exhaustivité. Deuxièmement, l'optimalité, en ce qui concerne les temps de réponse est atteinte par deux de ces méthodes, le serveur Earliest Deadline Late et la version optimale du serveur à utilisation totale.

Ces deux algorithmes ne sont pas implémentables à cause de leur surcoût en-ligne, mais leurs performances peuvent être approchées par le serveur à échange de priorités amélioré et la version améliorée du serveur à utilisation totale (TBS(k)). Le premier nécessitant le stockage en mémoire du vecteur des périodes d'inactivité sur l'hyperpériode, on lui préférera souvent le second [95].

La figure 6.3.5.3 page 175 permet de comparer les temps de réponse moyens obtenus par le serveur à scrutation dynamique (Polling), le serveur sporadique dynamique (DSS), le serveur à échange de priorités dynamique (DPE), le serveur à utilisation totale (TBS) et le serveur à échange de priorités dynamique amélioré (IPE).

6.3.5.4 Comparatif des différentes approches

Si les tâches périodiques ne possèdent que des contraintes d'échéance et l'algorithme d'ordonnancement n'est pas imposé, on a tout intérêt à utiliser Earliest Deadline First et la version améliorée du serveur à utilisation totale. Dans [104], une étude montre que les temps de réponse obtenus avec le slack stealing peuvent être améliorés avec seulement une ou deux itérations du serveur à utilisation totale (TBS(1) et TBS(2)). Or la complexité d'implémentation et le surcoût est bien moindre pour la version améliorée du serveur à utilisation totale que pour le slack stealing.

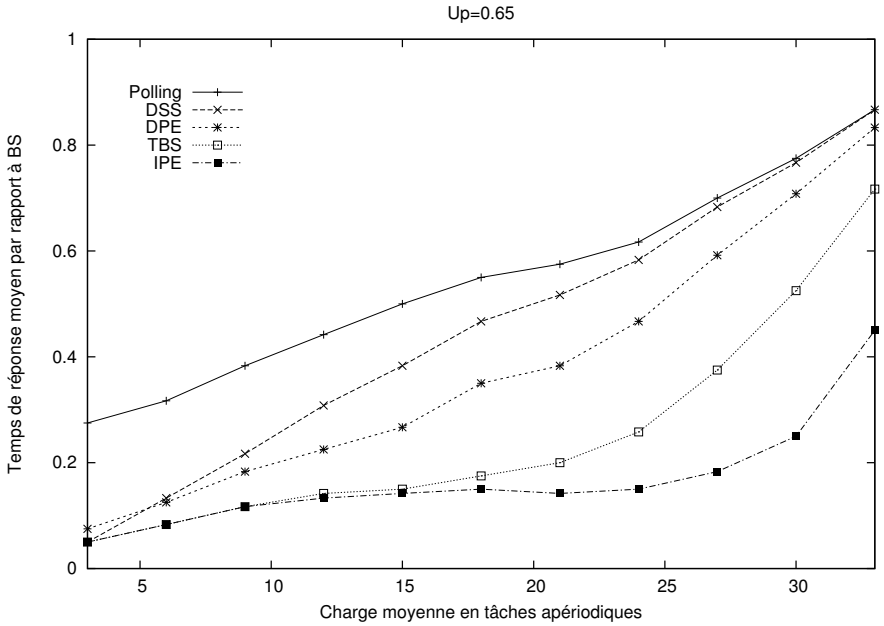

FIG. 6.17 – *Temps de réponse moyens relativement à BS en fonction de la charge due aux tâches apériodiques*

Le serveur à utilisation totale nécessite néanmoins que les échéances relatives soient égales aux périodes.

Pour ce qui est des approches hors-ligne, il n'existe à notre connaissance pas de meilleure solution que le slot shifting. Adaptable à différents types de contraintes et de tâches, il ne pose que le problème du coût mémoire qui croît avec l'hyperpériode.

On remarque que les approches dédiées aux tâches apériodiques fermes sont rares, et que la seule implémentable de par sa complexité est le slot shifting.

Chapitre 7

Contraintes de latence et ordonnancement mixte hors-ligne en-ligne

7.1 Limites du modèle temps réel classique

Le modèle temps réel classique tel que décrit précédemment (voir 6.1.2 page 138) a peu évolué depuis l'article de Liu et Layland. Bien sûr, d'autres grandeurs ont été définies, le modèle a été enrichi, mais le problème se résume toujours à un jeu de tâches indépendantes où la seule contrainte est l'échéance. C'est par exemple le cas quand on réduit un jeu de tâches avec contraintes de précédence à un jeu de tâches indépendantes avec échéances [105]. L'attachement à ce modèle tient principalement au fait qu'il permet l'utilisation d'algorithmes d'ordonnancement simples pouvant être implémentés sous la forme d'un ordonnanceur sans gros surcoût dans une approche en-ligne.

Ce modèle temps réel est utilisé par diverses communautés : traitement du signal, application client-serveur, lois de commande. Cette dernière communauté s'intéresse depuis des décennies à ce qu'on appelle le *Real-Time Control* (ici *control* veux dire loi de commande ou asservissement, et non pas contrôle au sens où nous l'entendons dans la première partie). Très tôt elle a pointé les insuffisances du modèle classique et surtout l'impact des ordonnancements l'utilisant sur les performances des lois de commande. Le fait qu'une tâche d'acquisition ne s'exécute pas périodiquement au sens strict du terme ($\forall j\, r_{i,j} = s_{i,j}$) peut, par exemple, dégrader la commande produite par le système. Cette communauté a alors décidé de travailler, non pas à améliorer le modèle mais à prendre en compte l'impact des ordonnancements dans les lois de commande. Cela a donné lieu à d'importants résultats dont le lecteur pourra trouver un panorama dans [135].

Ainsi, on a vu récemment émerger un nouvel axe de recherche qui vise à trouver des solutions d'ordonnancement pour les applications de type de loi de commande. Les algorithmes et heuristiques d'ordonnancement issus de leurs travaux manipulent des contraintes diverses et permettent de garantir un certain niveau de performance dans la loi de commande. Par exemple, un délai maximum entre acquisition et calcul de la commande peut-être requis, ceci afin de garantir une certaine "fraîcheur" des données traitées. En effet, il n'existe pas dans le modèle temps réel classique précédemment décrit de contraintes temporelles permettant de définir une relation entre les dates d'exécution de plusieurs tâches.

La seule contrainte issue du modèle classique permettant de contrôler la date de fin d'exécution d'une tâche est l'échéance. Mais cette date est soit fixe (échéance absolue) soit liée à la date d'activation de cette même tâche (échéance relative). La contrainte d'échéance ne permet pas de lier directement la fin d'exécution d'une tâche au début d'exécution d'une autre. Si on veut spécifier une telle contrainte liant le début d'exécution d'une tâche à la fin d'exécution d'une autre en utilisant le modèle classique, il faut modifier l'échéance de la deuxième tâche en fonction des caractéristiques de la première comme cela est fait par exemple dans [136].

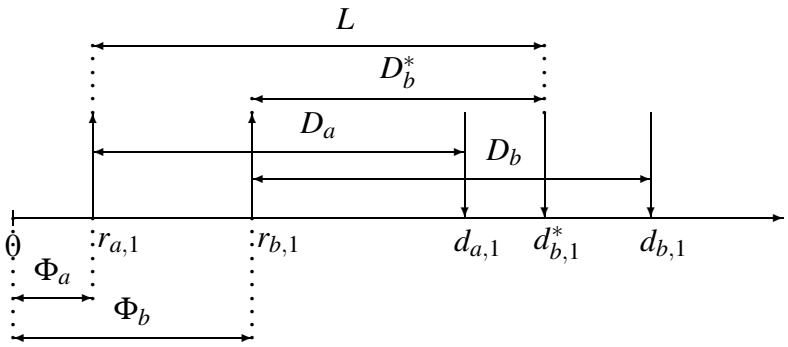

FIG. 7.1 – *Modification de l'échéance de T_b pour contraindre sa date de fin d'exécution par rapport à T_a*

Prenons l'exemple de deux tâches périodiques $T_a(\Phi_a,C_a,D_a,P_a)$ et $T_b(\Phi_b,C_b,D_b,P_b)$ ayant la même période, c'est-à-dire avec $P_a = P_b$. La loi de commande implémentée par ce couple de tâche nécessite que T_b finisse son exécution au plus tard L unités de temps après que T_a ait débuté la sienne. Il faut donc modifier l'échéance de T_b. Le lemme suivant donne la méthode pour déterminer cette échéance et prouve que la contrainte L est satisfaite. Ceci est illustré par la figure 7.1 page 178.

LEMME 2
Soit deux tâches $T_a(\Phi_a,C_a,D_a,P)$ et $T_b(\Phi_b,C_b,D_b,P)$ de même période P et une contrainte L définissant la durée maximale pouvant séparer le début d'exécution de T_a de la fin d'exécution de T_b. La contrainte L est satisfaite si l'échéance de T_b est modifié comme suit :

$$D_b^* = \min(D_b,(\Phi_a + L - \Phi_b)) \tag{7.1}$$

Les échéances absolues obtenues sont telles que :

$$d_{b,k}^* \leq \Phi_b + (k-1)P + \Phi_a + L - \Phi_b \tag{7.2}$$

C'est-à-dire :

$$d_{b,k}^* \leq (k-1)P + \Phi_a + L \tag{7.3}$$

PREUVE DU LEMME 2
On va montrer que si les instances successives de T_a et T_b respectent leurs échéances, alors la contrainte L est satisfaite, c'est-à-dire $\forall k\ f_{b,k} - s_{a,k} \leq L$.

Or $f_{b,k} - s_{a,k}$ est maximal quand $s_{a,k} = r_{a,k}$ et $f_{b,k} = d^*_{b,k}$. Le délai maximum entre le début d'exécution de T_a et la date de fin de T_b est alors :

$$\max_{k\in\mathbb{N}}(f_{b,k} - s_{a,k}) = \Phi_b + (k-1)P + D^*_b - (\Phi_a + (k-1)P) \tag{7.4}$$

$$\max_{k\in\mathbb{N}}(f_{b,k} - s_{a,k}) = \Phi_b - \Phi_a + D^*_b \tag{7.5}$$

Avec $D^*_b = \min(D_b,(\Phi_a + L - \Phi_b))$:

$$\max_{k\in\mathbb{N}}(f_{b,k} - s_{a,k}) \leq \Phi_b - \Phi_a + (\Phi_a + L - \Phi_b) \tag{7.6}$$

$$\max_{k\in\mathbb{N}}(f_{b,k} - s_{a,k}) \leq L \tag{7.7}$$

donc la contrainte L est satisfaite□

La réciproque de ce lemme est qu'un ordonnancement qui respecte la contrainte L respecte forcément l'échéance modifiée. Si cette réciproque n'est pas vraie cette modification de l'échéance restreint l'ensemble des ordonnancements corrects du jeu de tâches contenant T_a et T_b. On dit alors que la modification de l'échéance sur-contraint le système. C'est cette non-réciprocité que prouve le lemme suivant et du même coup la sur-contrainte du système par la modification d'échéance.

LEMME 3
La réciproque du lemme 2 n'est pas vraie.

PREUVE DU LEMME 3
Considérons le cas où l'instance de la tâche T_a commence son exécution le plus tard possible et satisfait son échéance. Cette date est donnée par $s_{a,k} = d_{a,k} - C_a$. Afin de satisfaire L, cela oblige T_b à finir son exécution avant la date $f_{b,k}$ définie comme suit :

$$f_{b,k} = d_{a,k} - C_a + L \tag{7.8}$$

$$f_{b,k} = r_{a,k} + D_a - C_i + L \tag{7.9}$$

$$f_{b,k} = \Phi_a + (k-1)P + D_a - C_a + L \tag{7.10}$$

Dans un jeu de tâche où $D_a > C_a$ (c'est le cas le plus souvent), il peut donc exister un ordonnancement qui satisfasse L mais dans lequel $f_{b,k} > d^*_{b,k}$, c'est-à-dire que l'échéance modifiée ne soit pas satisfaite □

Lorsque le système est sur-contraint, cela peut mener à l'impossibilité de trouver un ordonnancement correct pour un jeu de tâches comportant une contrainte L. Prenons l'exemple de 2 tâches $T_1(0,2,10,10)$ et $T_2(0,3,10,10)$, et une contrainte L indiquant qu'au plus 6 unités de temps peuvent séparer le début d'exécution de T_1 de la fin de T_2. Si on modifie l'échéance de T_2 comme indiqué précédemment, on obtient $T^*_2(0,3,6,10)$. Si le jeu de tâches comporte une troisième tâche $T_3(0,8,20,20)$ et que l'on choisit d'ordonnancer sans préemption, il n'existe pas d'ordonnancement correct pour le jeu de tâche T_1, T^*_2, T_3. Or il existe un ordonnancement correct satisfaisant L pour le jeu de tâche T_1, T_2, T_3, comme le montre la figure 7.2 page 180.

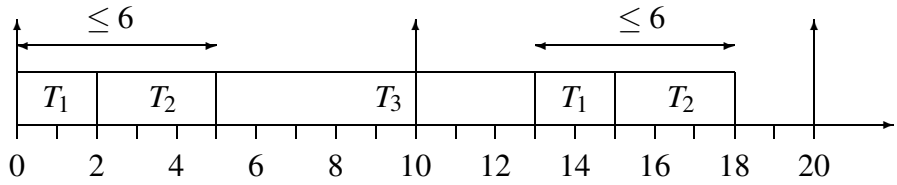

FIG. 7.2 – *Exemple d'ordonnancement de T_1, T_2, T_3 respectant $\mathcal{C}$*

7.2 Définition de la contraintes de latence

Il y eu peu de travaux sur ce type de contrainte et les travaux s'en approchant utilisent des termes différents les uns des autres. Une définition claire et générique de cette contrainte a été donnée dans la thèse [137] et exploitée dans l'article [138].

DÉFINITION 1
On appelle $L(T_a,T_b)$ la contrainte de latence liant les tâches T_a et T_b, et L_{ab}, $L_{ab} \in \mathbb{N}$ la durée spécifiée pour cette contrainte. $L(T_a,T_b) = L_{ab}$ implique :*

- *$P_a = P_b$, les deux tâches ont la même période,*
- *$T_a = T_b$ ou $T_a \rightarrow T_b$, c'est-à-dire que $\forall k \quad f_{a,k} \leq s_{b,k}$,*
- *$\forall k, \quad f_{b,k} \leq s_{a,k} + L_{ab}$.*

On étend ici la définition originale au cas où $T_a = T_b$. De plus on impose $P_a = P_b$ car le cas échéant il est impossible de trouver un ordonnancement correct.

La contrainte de latence est une extension des contraintes bout-en-bout (*end-to-end*) [139]. Une contrainte bout-en-bout définit la durée maximum tolérée pour l'exécution d'un graphe de précédence. Contrairement à la contrainte de latence, cette contrainte ne concerne pas un couple précis de tâches dans le graphe, ce qui empêche de spécifier plusieurs contraintes bout-en-bout pour une même composante connexe du graphe. On peut malgré tout spécifier une contrainte bout-en-bout par composante connexe. Les auteurs de [140] montrent comment déduire de ce type de contrainte, ainsi que d'autres contraintes complexes, les caractéristiques du jeu de tâche relativement au modèle classique. Une fois les périodes et échéances calculées, le jeu de tâches est ordonnancé avec les algorithmes d'ordonnancement classiques (DM, EDF). [141] est consacré à l'ordonnancement non préemptif de tâches avec contraintes de précédence et de latence. Les auteurs y présentent des algorithmes d'ordonnancement et des conditions d'ordonnançabilité.

Ainsi, contrairement aux contraintes bout-en-bout, on peut spécifier plusieurs contraintes sur une même composante connexe du graphe. Une contrainte de latence peut même s'appliquer à une seule tâche, si on définit $L(T_a,T_a)$. Dans le cas préemptif cette contrainte permet ainsi de limiter la durée totale de préemption d'une tâche. Une autre application possible concerne les sections critiques. Soit T une tâche de faible priorité et de grande période mais effectuant un long accès à une ressource critique. Si cette tâche est préemptée trop longtemps, l'accès à la ressource critique va se prolonger et risque de bloquer d'autres tâches utilisant cette ressource. Plutôt que d'utiliser

une échéance qui forcerait la tâche à s'exécuter au début de sa période, ont peut définir une latence $L(T_a,T_a)$ qui bornera la durée de l'accès à la ressource. En effet on aura alors $\forall k, f_{a,k} - s_{a,k} \leq L_{aa}$.

7.3 Problème à résoudre

Le problème que nous cherchons à résoudre est le suivant. Un jeu de tâches périodiques avec des contraintes d'échéances, de précédence et de latence a été ordonnancé hors-ligne sur une architecture distribuée. Nous ne nous intéressons pas aux algorithmes ou heuristiques employés, ceux-ci pouvant très bien sur-contraindre le jeu de tâches (transformation des latences en échéances) pour réussir à l'ordonnancer comme dans [136], ou ordonnancer ces tâches directement mais dans un modèle où la périodicité est stricte ($\forall i,ks_{i,k} = r_{i,k}$) comme dans [141].

Le résultat est composé de p ordonnancements hors-ligne, où p est le nombre de processeurs [1]. Chaque ordonnancement hors-ligne est une séquence de tâches $(T_1,T_2,T_3,\ldots,T_n)$ où $i < j$ implique que T_i précède T_j dans l'ordonnancement et T_1 et T_n sont respectivement la première et la dernière tâche à exécuter. Chaque ordonnancement hors-ligne définit l'ordre d'exécution des tâches sur l'hyperpériode (définition au 6.1.3 page 141). En-ligne, cet ordre ne pourra pas être modifié.

Nous avons vu, dans la première partie de la thèse que dans les ordonnancements de programmes conditionnés, deux tâches (ou plus) alternatives pouvaient être ordonnancées dans le même intervalle de temps. La gestion de ces alternatives compliquant le problème, nous considérons ici qu'un ordonnancement contenant des alternatives aura été préalablement transformé en un ordonnancement n'en comportant pas. Cette transformation parcourt l'ordonnancement initial et ses alternatives en choisissant à chaque fois la tâche qui utilise le processeur le plus longtemps et les contraintes d'échéances et de latence les plus contraignantes. L'ordonnancement alors obtenu est une version pessimiste de l'ordonnancement initial.

Il est à noter que couvrant l'hyperpériode, l'ordonnancement hors-ligne est composé en réalité d'une suite d'instances de tâches et non pas d'une suite de tâches. Néanmoins une fois les tâches ordonnancées et les périodicités respectées, l'information propre aux instances (appartenance à une même tâche) n'est plus utile et surcharge la notation. On utilisera donc le terme tâche pour désigner chaque élément de la séquence d'exécution définie par l'ordonnancement hors-ligne. Il ne faut pas oublier néanmoins que le terme ordonnancement hors-ligne s'applique exclusivement aux tâches périodiques.

Il est possible que les algorithmes ou heuristiques utilisés pour calculer les ordonnancements hors-ligne soient préemptifs. Dans ce cas un ordonnancement hors-ligne fixe les dates de préemption. Cela permet d'abord de prendre en compte le coût de la préemption : on sait quelle tâche préempte quelle autre tâche et donc la taille du contexte à sauvegarder. De plus un tel ordonnancement est totalement déterministe car il garantit que si une tâche termine son exécution plus tôt que prévu (si l'instance a une durée d'exécution inférieure au Worst Case Execution Time), les dates et l'ordre d'exécution des tâches suivantes resteront inchangés. Pour cela, la décision d'ordonnancement préemptif "T_1 commence son exécution, T_2 préempte $T1$ puis $T1$ termine son exécution" apparaît dans l'ordonnancement sous la forme de trois tâches ($T_{1'}$, $T_{2'}$, $T_{3'}$) dont les dates

1. Bien sûr il existe aussi des ordonnancements des dépendances de données sur les média de communications, mais ils ne seront d'aucune utilité ici.

d'exécution correspondent à ce scénario de préemption. En résumé, les tâches qui composent un ordonnancement hors-ligne ne correspondent pas exactement au jeu de tâches initial. Elles en sont des instances, voir des parties d'instances, à exécuter au cours de l'hyperpériode. Leurs caractéristiques sont cependant directement liées aux caractéristiques des tâches initiales ainsi qu'aux décisions d'ordonnancement et de préemption.

On veut pouvoir exécuter en-ligne, sur chaque processeur des tâches apériodiques souples ou fermes tout en respectant les contraintes de latence et d'échéance. On résoudra ce problème localement pour chaque processeur, la migration de tâches n'étant pas permise.

L'exécution de tâches apériodiques supplémentaires peut modifier les dates d'exécution des tâches ordonnancées hors-ligne. On devra veiller à ce que les contraintes satisfaites par l'ordonnancement hors-ligne le restent en-ligne. A ce titre, un ordonnancement hors-ligne indique aussi pour chaque tâche son échéance absolue [2] et sa contrainte de latence si elle en possède une.

A chaque tâche T_i de l'ordonnancement hors-ligne est donc associée la ou les contraintes la concernant, à savoir son échéance et/ou sa contrainte de latence. Les échéances sont absolues. De plus, chaque tâche T_i possède une date de début au plus tôt* notée est_i (*earliest start time*) qui peut être supérieure à la date de fin d'exécution de la tâche précédente si la tâche ne peut pas être active directement après la fin de celle qui la précède. Elle ne peut pas, par contre, être inférieure à la date de fin de la tâche la précédant dans l'ordonnancement.

En ce qui concerne les communications, on considère que les dates auxquelles elles ont lieu ne peuvent être modifiées. En effet, traitant le problème de façon locale, on ne peut prévoir les conséquences sur le processeur destination d'une communication repoussée par le processeur source. Ainsi, pour une tâche productrice d'une donnée à envoyer, la date de l'envoi de cette donnée correspondra à son échéance (à moins que cette tâche possède elle même une échéance plus petite). De même, une tâche T_i recevant une donnée via une communication ne pourra avoir un est_i plus petit que la date de fin de la communication.

Il est avéré que le problème d'ordonnancement de tâches périodiques avec contraintes de latence est plus complexe [137] que l'ordonnancement de tâches périodiques avec contraintes d'échéance. Néanmoins, en ce qui concerne l'exécution en-ligne de tâches apériodiques, nous montrerons que ce type de contrainte offre un degré de liberté supplémentaire permettant d'augmenter les performances tant au niveau des temps de réponses des tâches apériodiques souples que de l'acceptation des tâches apériodiques fermes. Nous verrons également sous quelles conditions ce degré de liberté peut être exploité avec une faible complexité afin d'être applicable dans le contexte en-ligne.

7.4 Réduction de l'ensemble des contraintes à respecter

7.4.1 Principes

Chaque ordonnancement hors-ligne σ satisfait un ensemble noté E_σ de contraintes de latence et d'échéance. Lorsque en-ligne des tâches apériodiques viendront s'insérer dans la séquence de

2. Une tâches ne peut avoir qu'une contrainte de latence, mais sa périodicité implicitement définit une échéance du type $D_i = P_i$.

tâches définie par σ, il faudra veiller à ce que les contraintes composant E_σ restent satisfaites. Parce que l'ordonnancement hors-ligne σ définit un ordre d'exécution inviolable des tâches concernées par ces contraintes, nous allons montrer que l'on peut trouver un ensemble de contraintes E'_σ tel que $E'_\sigma \subset E_\sigma$ et que, si toutes les contraintes appartenant à E'_σ sont satisfaites, alors toutes les contraintes appartenant à E_σ le sont également. On dira alors que le respect des contraintes de E_σ est garanti par le respect des contraintes de E'_σ.

La complexité des calculs à effectuer en-ligne pour exécuter des tâches apériodiques étant directement liée aux nombres de contraintes à satisfaire, cette réduction de l'ensemble des contraintes permet de diminuer les calculs et donc le surcoût en-ligne.

Nous rappellerons, dans un premier temps, comment l'ordre défini par l'ordonnancement hors-ligne permet de réduire le nombre des valeurs différentes que peuvent prendre les échéances absolues. Puis, nous donnerons une condition sous laquelle le respect d'une contrainte de latence est garanti par le respect d'une échéance. Ensuite, nous présenterons une condition sous laquelle le respect d'une contrainte de latence est garanti par le respect d'une autre contrainte de latence. Nous élargirons cette condition au cas où une contrainte de latence est garantie par le respect d'un ensemble de contraintes de latence. Enfin, nous présenterons un exemple d'ordonnancement hors-ligne auquel nous appliquerons cette réduction de l'ensemble des contraintes à respecter.

7.4.2 Uniformisation des échéances

En s'appuyant sur le fait que l'ordre d'exécution des instances des tâches périodiques est défini par l'ordonnancement hors-ligne et ne peut être modifié, on peut modifier les échéances des tâches de manière à réduire le nombre de valeurs différentes d'échéances (certaines tâches ont alors la même échéance) comme cela est fait dans [2]. Cette modification appliquée à l'ensemble d'un ordonnancement σ est appelée uniformisation des échéances* de l'ordonnancement σ.

Soit une tâche T_i appartenant à un ordonnancement hors-ligne σ. Pour que T_{i+1} respecte son échéance, T_i (qui doit être exécutée avant T_{i+1}) ne peut terminer son exécution après d_{i+1}. On peut définir pour T_i une nouvelle échéance d_i^* telle que :

$$d_i^* = \min(d_i, d_{i+1}) \tag{7.11}$$

En appliquant l'équation 1, en partant de la fin de l'ordonnancement hors-ligne (de T_n à T_1), on obtient un ordonnancement σ' tel que :

- les échéances sont croissantes de d_1 à d_n,
- le nombre des valeurs différentes prises par les échéances dans σ' est inférieur ou égal à celui de σ.

Par la suite, on considère que les échéances des tâches de l'ordonnancement ont été modifiées selon cette méthode.

7.4.3 Contrainte de latence respectée si une échéance est respectée

Les tâches possédant une contrainte de latence n'en ont pas moins une échéance. Si l'on considère $L(T_i, T_j)$, il se peut que le respect par T_j de son échéance d_j implique que la contrainte de

latence soit respectée. D'où ce premier théorème :

THÉORÈME 7.1
Soit une contrainte de latence $L(T_i,T_j)$. Si $d_j \leq est_i + L_{ij}$, alors le respect de $L(T_i,T_j)$ est garanti par le respect de d_j.

Preuve : Raisonnement par contradiction. Supposons que l'échéance d_j soit respectée mais que $L(T_i,T_j)$ ne le soit pas. Si $L(T_i,T_j)$ n'est pas respectée, $s_i + L_{ij} < f_j$ et comme $s_i \geq est_i$, nous avons $est_i + L_{ij} < f_j$. Or par définition $d_j \leq est_i + L_{ij}$ ce qui implique $d_j < f_j$ ce qui est en contradiction avec le fait que l'échéance d_j soit respectée et prouve le théorème ⊡

7.4.4 Contrainte de latence respectée si une autre contrainte de latence est respectée

Considérons une contrainte de latence $L(T_i,T_j)$ et appelons "intervalle de la latence $L(T_i,T_j)$", l'intervalle $[s_i,f_j]$. Soit un ordonnancement σ et une deuxième contrainte de latence $L(T_g,T_h)$, telle T_g, T_h, T_i et T_j appartiennent à cet ordonnancement. Suivant l'ordre d'exécution des tâches défini par σ, les intervalles des latences $L(T_i,T_j)$ et $L(T_g,T_h)$:

- sont disjoints si $j < g$ ou $h < i$,
- sont en inclusion* si $i \leq g \leq h \leq j$ (intervalle de latence de $L(T_g,T_h)$ inclus dans l'intervalle de latence $L(T_i,T_j)$) ou si $g \leq i \leq j \leq h$ (intervalle de latence de $L(T_i,T_j)$ inclus dans l'intervalle de latence $L(T_g,T_h)$),
- se chevauchent* si $i < g \leq j < h$ ou $g < i \leq h < j$.

Dans le cas d'inclusion, la contrainte de latence correspondant à l'intervalle inclus peut être garantie par le respect de la contrainte de latence correspondant à l'intervalle englobant.

THÉORÈME 7.2
Soit $L(T_g,T_h)$ et $L(T_i,T_j)$ deux contraintes de latences tel que dans l'ordonnancement hors-ligne σ il y a inclusion de l'intervalle de latence de $L(T_i,T_j)$ dans l'intervalle de latence de $L(T_g,T_h)$. Le respect de $L(T_i,T_j)$ est garanti par le respect de la contrainte de latence $L(T_g,T_h)$ si :

$$L_{gh} - \sum_{k=g}^{h} C_k \leq L_{ij} - \sum_{m=i}^{j} C_m \tag{7.12}$$

Preuve : Raisonnement par contradiction. Considérons que la condition (7.12) est remplie et que la contrainte de latence $L(T_g,T_h)$ est respectée alors que la contrainte de latence $L(T_i,T_j)$ ne l'est pas.

Notons A_{pq} la durée cumulée, entre le début d'exécution d'une tâche T_p et la fin d'exécution d'une tâche T_q, des intervalles de temps durant lesquelles le processeur n'exécute pas une tâche appartenant à σ[3]. A_{pq} peut être calculé pour n'importe quel couple de tâche (T_p,T_q) par la formule

3. Il ne s'agit pas forcément de temps d'inactivité processeur. En effet, il ne faut pas oublier qu'en-ligne des tâches apériodiques viendront s'ajouter à celles de σ, ce qui pourra repousser les dates d'exécution de ces dernières.

suivante :

$$A_{pq} = (f_q - s_p) - \sum_{k=p}^{q} C_k \tag{7.13}$$

Puisque la contrainte de latence $L(T_g,T_h)$ est respectée, on a $f_h - s_g \leq L_{gh}$. Cela nous permet de borner A_{gh} :

$$A_{gh} \leq L_{gh} - \sum_{k=g}^{h} C_k \tag{7.14}$$

Au contraire, la contrainte de latence $L(T_i,T_j)$ n'est, par hypothèse, pas respectée, ce qui signifie que $f_j - s_i > L_{ij}$ et ainsi :

$$A_{ij} > L_{ij} - \sum_{k=i}^{j} C_k \tag{7.15}$$

Les intervalles de latence étant en inclusion, T_g est exécuté avant T_i et T_j avant T_h. La durée correspondant à A_{ij} est donc incluse dans A_{gh} et par conséquence $A_{ij} \leq A_{gh}$. Ceci ajouté à (7.14) et (7.15) entraîne :

$$L_{gh} - \sum_{k=g}^{h} C_k \geq A_{gh} \geq A_{ij} > L_{ij} - \sum_{k=i}^{j} C_k \tag{7.16}$$

ce qui est en contradiction avec (7.12) et prouve le théorème ⊡

7.4.5 Contrainte de latence respectée si d'autres contraintes de latence sont respectées

Ce dernier résultat est une extension du précédent au cas où il y a inclusion d'un intervalle de latence dans un intervalle composé de plusieurs intervalles de latence se chevauchant. Un tel intervalle correspond à une séquence de contraintes de latence où pour toute paire de contraintes de latence $[L(T_a,T_b),L(T_c,T_d)]$ consécutives dans la séquence, l'ordonnancement σ est tel que $a < c \leq b < d$.

THÉORÈME 7.3
Soit $L(T_i,T_j)$ une contrainte de latence et $O = [L(T_a,T_b),\ldots,L(T_v,T_w)]$ définissant une séquence d'intervalles de latence se chevauchant telle que $a \leq i \leq j \leq w$.

La contrainte de latence $L(T_i,T_j)$ est garantie par le respect de toutes les contraintes appartenant à O si :

$$\sum_{L(T_p,T_q)\in O} \left(L_{pq} - \sum_{k=p}^{q} C_k \right) \leq L_{ij} - \sum_{m=i}^{j} C_m \tag{7.17}$$

Preuve : Raisonnement par contradiction. Considérons que la condition (7.17) est remplie, que l'ensemble des contraintes de latence appartenant à O sont respectées alors que la contrainte de latence $L(T_i,T_j)$ ne l'est pas.

Si on utilise $A_{ij} = (f_j - s_i) - \sum_{k=i}^{j} C_k$ défini précédemment, le non-respect de la contrainte de latence $L(T_i,T_j)$ implique :

$$A_{ij} > L_{ij} - \sum_{k=i}^{j} C_k \tag{7.18}$$

Au contraire pour chaque contrainte de latence $L(T_p,T_q) \in O$, on a :

$$A_{pq} \leq L_{pq} - \sum_{k=p}^{q} C_k \tag{7.19}$$

Comme $a \leq i \leq j \leq w$, on a inclusion de A_{ij} dans A_{aw} et donc $A_{ij} \leq A_{aw}$. En utilisant O, la séquence de contraintes de latence, on obtient :

$$A_{ij} \leq A_{aw} \leq \sum_{L(T_p,T_q)\in O} A_{pq} \leq \sum_{L(T_p,T_q)\in O} \left(L_{pq} - \sum_{k=p}^{q} C_k \right) \tag{7.20}$$

En utilisant (7.20) et (7.18), on a finalement :

$$L_{ij} - \sum_{k=i}^{j} C_k < A_{ij} \leq \sum_{L(T_p,T_q)\in O} \left(L_{pq} - \sum_{k=p}^{q} C_k \right) \tag{7.21}$$

ce qui est en contradiction avec (7.17) et prouve le théorème ⊡

7.4.6 Exemple

Nous allons utiliser les résultats précédents afin de réduire le nombre de contraintes sur un exemple. La figure 7.3 page 186 montre un ordonnancement hors-ligne.

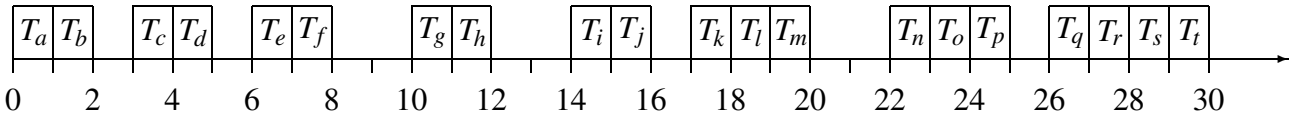

FIG. 7.3 – *Exemple d'ordonnancement satisfaisant des contraintes de latence*

Les dates d'exécution au plus tôt des tâches (est_x) ainsi que leurs échéances sont données dans le tableau 7.2 page 187. Des contraintes de latences sont respectées par cet ordonnancement. Elles sont au nombre de 7 et sont données par le tableau 7.1 page 186.

$L(T_x,T_y)$	$L(T_a,T_f)$	$L(T_b,T_d)$	$L(T_g,T_s)$	$L(T_i,T_m)$	$L(T_j,T_r)$	$L(T_l,T_p)$	$L(T_n,T_t)$
L_{xy}	10	8	23	7	16	7	8

TAB. 7.1 – *Contraintes de latence sur l'ordonnancement de la figure 7.3 page 186*

T_x	T_a	T_b	T_c	T_d	T_e	T_f	T_g	T_h	T_i	T_j
est_x	0	1	3	4	6	7	10	11	14	15
d_x	11	12	13	12	15	11	35	15	24	34
d_x^*	11	11	11	11	11	11	15	15	24	24
T_x	T_k	T_l	T_m	T_n	T_o	T_p	T_q	T_r	T_s	T_t
est_x	17	18	19	22	23	24	26	27	28	29
d_x	26	33	24	32	40	33	40	34	35	32
d_x^*	24	24	24	32	32	32	32	32	32	32

TAB. 7.2 – *Dates de début au plus tôt, échéances et échéances modifiées des tâches de l'ordonnancement de la figure 7.3 page 186*

La première étape consiste à appliquer l'équation 7.11 page 183 pour uniformiser les échéances. En remontant l'ordonnancement hors-ligne, de la tâche T_t à la tâche T_a, on obtient les échéances notées d_x^* dans le tableau 7.2 page 187. On réduit ainsi le nombres d'échéances différentes de 11 à 4.

La deuxième étape consiste à supprimer les contraintes de latence garanties par le respect d'une échéance. En appliquant le résultat du théorème 7.1 page 184, on peut supprimer $L(T_g,T_s)$ car on a $d_s^* = 32$ et $est_g + L_{gs} = 10+23 = 33$ donc $d_s^* \leq est_g + L_{gs}$.

La troisième étape utilise le résultat du théorème 7.2 page 184 pour supprimer les contraintes de latence garanties par le respect d'une autre contrainte de latence par inclusion. Ici, on peut supprimer $L(T_b,T_d)$ car on a :

$$L_{af} - \sum_{x=a}^{f} C_x \leq L_{bd} - \sum_{y=b}^{d} C_y \quad \text{car} \quad \begin{cases} L_{af} - \sum_{x=a}^{f} C_x = 10-6 = 4 \\ L_{bd} - \sum_{y=b}^{d} C_y = 8-5 = 3 \end{cases} \tag{7.22}$$

Enfin, la dernière étape utilise le résultat du théorème 7.3 page 185 pour supprimer les contraintes de latence garanties par le respect d'autres contraintes de latence par inclusion dans l'intervalle correspondant à une séquence de contraintes de latences dont les intervalles se chevauchent. Ici, on peut supprimer $L(T_j,T_r)$ car on a :

$$\begin{cases} L_{im} - \sum_{w=i}^{m} C_w = 7-5 = 2 \\ L_{lp} - \sum_{x=l}^{p} C_x = 7-5 = 2 \\ L_{nt} - \sum_{y=n}^{t} C_y = 8-7 = 1 \\ L_{jr} - \sum_{z=j}^{r} C_z = 16-9 = 7 \end{cases} \quad \text{et donc} \tag{7.23}$$

$$\left(L_{im} - \sum_{w=i}^{m} C_w\right) + \left(L_{lp} - \sum_{x=l}^{p} C_x\right) + \left(L_{nt} - \sum_{y=n}^{t} C_y\right) \leq L_{jr} - \sum_{z=j}^{r} C_z \tag{7.24}$$

Il suffit donc de respecter les contraintes de latence $L(T_a,T_f)$, $L(T_i,T_m)$, $L(T_l,T_p)$ et $L(T_n,T_t)$ ainsi que les échéances de T_f, T_h, T_m et T_t pour garantir le respect de l'ensemble des contraintes (échéance et latence) des tâches composant l'ordonnancement hors-ligne.

Chapitre 8

Ordonnancement temps réel mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques

8.1 Choix de la méthode

Parmi les méthodes présentées dans l'état de l'art, le slot shifting (voir 6.3.3.1 page 166) semble naturellement le plus adapté à notre cas, car il est le seul à reposer sur un ordonnancement hors-ligne des tâches périodiques. Cet argument seul ne suffit pas, car on pourrait très bien convertir notre ordonnancement hors-ligne en un ordonnancement en-ligne à priorités fixes, en utilisant par exemple la méthode présentée dans [142]. De même, une fois les échéances uniformisées comme expliqué dans le chapitre précédent et les dates de début au plus tôt modifiées par l'algorithme présenté dans [105] (voir 6.1.4 page 145), on pourrait très bien utiliser un ordonnanceur de type Earliest Deadline First.

L'approche consistant à passer par un ordonnancement à priorités fixes est à écarter car elle ne permet pas d'atteindre l'optimalité en ce qui concerne les temps de réponse [133] (voir 6.3.5.1 page 169). De plus, les approches de type serveur reposent sur le fait que l'on puisse, après étude d'ordonnançabilité des tâches périodiques, rajouter facilement une tâche, c'est à dire calculer le rapport C/P du serveur grâce à une condition d'ordonnançabilité. Dans le cas d'ordonnancement hors-ligne sur architecture distribuée hétérogène (vitesse d'exécution différente sur des processeurs différents) avec prise en compte des communications, il n'existe pas de condition d'ordonnançabilité robuste. Si rajouter un serveur après ordonnancement hors-ligne sur chaque processeur n'est pas impossible, son dimensionnement est hasardeux.

En ce qui concerne un ordonnancement par Earliest Deadline First après uniformisation des échéances, se pose également le problème de définir les caractéristiques du serveur. En effet, elles se définissent habituellement grâce au facteur d'utilisation processeur, Earliest Deadline First étant optimal et permettant d'atteindre les 100% d'utilisation. Or, il semble impossible de prendre en

compte les contraintes de latence dans le calcul de ces facteurs d'utilisation, notamment dans le cas de contraintes de latence en inclusion ou se chevauchant [1]. De plus, puisqu'on relâche ici l'hypothèse simplificatrices des échéances égales aux périodes, le serveur à utilisation totale n'est plus applicable. En outre nous avons vu que peu de méthodes basées sur Earliest Deadline First offraient la possibilité d'accepter des tâches apériodiques fermes et qu'il n'y en avait pas dont la complexité fut acceptable en vu d'implantation. Cela est principalement dû au fait que ces techniques n'utilisent pas la connaissance avant exécution de l'ordonnancement pour limiter les calculs en-ligne.

Le slot shifting utilise la connaissance de l'ordonnancement ainsi que de la distribution (en prenant en compte les communications inter-processeurs). En outre, grâce au modèle de slot que nous détaillerons plus loin, il permet de prendre en compte le coût des préemptions des tâches périodiques par des tâches apériodiques. C'est pourquoi nous avons choisi de nous appuyer sur la méthode de slot shifting et d'y introduire le concept de latence. Nous allons d'abord décrire les principes du slot shifting de manière plus détaillée que dans l'état de l'art. Ces principes sont ceux de base présentés dans [125] et plus longuement dans la thèse [2].

8.2 Principes du slot shifting

Nous allons dans un premier temps préciser le modèle temporel ainsi que le modèle de tâche auquel s'applique le slot shifting. Ensuite, la description du slot shifting se fera en deux parties. La première partie concerne les calculs à effectuer, hors-ligne, sur l'ordonnancement des tâches périodiques. La deuxième partie détaille comment les tâches apériodiques peuvent être acceptées et exécutées en-ligne.

8.2.1 Modèles

Un modèle de temps discret est utilisé. Le temps est divisé en slot* par une horloge globale d'une granularité $slotlength$. Ces slots sont indexés temporellement de 0 à l'infini. L'intervalle de temps correspondant à une slot i, c'est-à-dire compris entre le début et la fin de cet intervalle est défini par $[slotlength * i, slotlength * (i+1)]$. Les slots sont de même taille sur tous les processeurs et commencent et finissent au mêmes dates. Le slot est l'unité de temps indivisible lors de l'exécution en-ligne, c'est-à-dire que s'il y a préemption d'une tâche, elle ne peut avoir lieu qu'entre deux slots à une date $t = slotlength * k$ où k est un entier positif. Les périodes des tâches ainsi que leurs échéances doivent donc être des multiples de $slotlength$.

Puisque les slots définissent en-ligne les dates possibles de préemption, il est possible sur un intervalle de durée k de borner le nombre de préemptions et de prendre en compte son coût dans le calcul des C_i du jeu de tâches. Il est également possible d'inclure dans la durée du slot, le coût des calculs en-ligne.

En ce qui concerne le modèle de tâche, chaque tâche appartient à une composante connexe du graphe de précédence appelée CCGP. Chaque CCGP possède une période et une échéance, dont

1. Le problème se pose dès qu'une tâche est impliquée dans deux contraintes de latence.

héritent les tâches qui la composent. On peut voir cette échéance comme une façon de définir une contrainte bout-en-bout (mais en sur-contraignant le jeu de tâches).

8.2.2 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques

8.2.2.1 Dates de début au plus tôt et échéances

Chaque ordonnancement hors-ligne σ définit pour chaque tâche une date de début au plus tôt, et une échéance. La date de début au plus tôt, notée est_i (comme définie dans la section 7.3 page 181), correspond soit à la périodicité de la CCGP à laquelle appartient la tâche, soit à la date de réception d'une donnée (communication venant d'un autre processeur) nécessaire à son exécution. De même, l'échéance d'une tâche T_i correspond soit à l'échéance de la CCGP à laquelle appartient la tâche, soit à la date de l'envoi d'une donnée (communication) produite par la tâche. En effet en-ligne, afin de garder la synchronisation inter-processeurs, les dates des communications ne peuvent être repoussées. En conséquence, quand une tâche produit une communication, la date d'envoi prévue est une échéance.

La premier calcul effectué hors-ligne sur l'ordonnancement σ est la modification des échéances telle que nous l'avons détaillée au paragraphe 7.4.2 page 183.

8.2.2.2 Intervalles d'exécution et spare capacities

Nous avons vu que cette modification des échéances pouvait conduire plusieurs tâches consécutives à posséder la même échéance (voir par exemple le tableau 7.2 page 187). Soit m le nombre d'échéances différentes que comporte l'ordonnancement σ. Ces échéances absolues découpent l'hyperpériode en intervalles disjoints appelés intervalles d'exécution*. Chaque échéance $d_i, 1 \leq i \leq m$ définit la fin d'un intervalle d'exécution et le début d'un autre. Chaque intervalle d'exécution $I_i\, 1 \leq i \leq m$ est défini par $]start(I_i), end(I_i)]$ où :

$$end(I_i) = d_i \quad \text{et} \quad start(I_i) = \begin{cases} d_{i-1} & \text{si } i \neq 1 \\ 0 & \text{sinon} \end{cases} \tag{8.1}$$

Chaque tâche T_k appartient à l'intervalle d'exécution I_i tel que $end(I_i) = d_k$. Les tâches ayant la même échéance appartiennent donc au même intervalle d'exécution. On note $|I_i|$ la taille de l'intervalle d'exécution I_i, c'est-à-dire la valeur que prend $end(I_i) - start(I_i)$.

Il est important de ne pas confondre intervalle d'exécution avec la notion de fenêtre d'exécution, parfois employée dans la littérature. La fenêtre d'exécution d'une tâche désigne l'intervalle compris entre sa date de début au plus tôt et son échéance. Une différence fondamentale est que les fenêtres d'exécution peuvent se chevaucher alors que les intervalles d'exécution sont, par définition, disjoints. Une seconde différence est que si une tâche s'exécute forcément, par définition, dans sa fenêtre d'exécution, il n'en est pas de même avec son intervalle d'exécution. En effet, la seule contrainte est qu'une tâche ne peut pas s'exécuter après la fin de son intervalle d'exécution, puisque la borne à droite de l'intervalle d'exécution correspond à l'échéance de la tâche.

Par contre une tâche peut débuter son exécution ou s'exécuter totalement avant le début de son intervalle d'exécution.

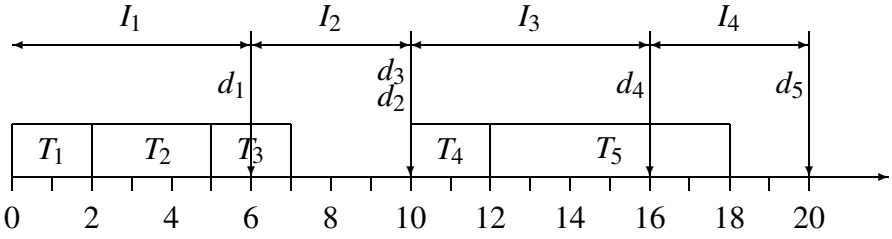

FIG. 8.1 – *Exemple de découpage d'ordonnancement en intervalles d'exécution*

Considérons l'ordonnancement σ constitué des tâches T_1, T_2, T_3, T_4 et T_5 où $C_1 = C_3 = C_4 = 2$, $C_2 = 3$ et $C_5 = 6$. Les tâches possèdent les échéances suivantes : $d_1 = 6$, $d_2 = d_3 = 10$, $d_4 = 16$ et $d_5 = 20$. Comme l'illustre la figure 8.1 page 192, l'ordonnancement σ est découpé en 4 intervalles d'exécution, $I_1[0,d_1]$, $I_2]d_1,d_3]$, $I_3]d_3,d_4]$ et $I_4]d_4,d_5]$ tels que $T_1 \in I_1$, T_2,$T_3 \in I_2$, $T_4 \in I_3$ et $T_5 \in I_4$. On remarque que T_1 et T_4 sont entièrement exécutées dans leurs intervalles d'exécution alors que T_2, T_3 et T_5 ne le sont pas.

On associe à chaque intervalle d'exécution sa spare capacity* comme étant le temps disponible dans cet intervalle pour exécuter des tâches apériodiques. Si toutes les tâches étaient exécutées dans leur intervalle d'exécution, il suffirait pour obtenir la spare capacity $sc(I_i)$ d'un intervalle d'exécution I_i, de retrancher à la taille de cet intervalle les durées d'exécution des tâches lui appartenant. Ce qui correspondrait à la formule :

$$sc(I_i) = |I_i| - \sum_{T_j \in I_i} C_j \tag{8.2}$$

Cependant, l'exemple précédent (figure 8.1 page 192) montre bien que les tâches ne s'exécutent pas forcément dans leur intervalle d'exécution. On doit donc prendre en compte, dans la spare capacity d'un intervalle d'exécution I_i, le temps utilisé à l'exécution de tâches appartenant aux intervalles d'exécution I_k avec $k > i$. C'est pourquoi les spare capacities des intervalles d'exécution sont calculées, hors-ligne, de I_m à I_1 grâce à la formule suivante :

$$sc(I_i) = \begin{cases} |I_i| - \sum_{T_j \in I_i} C_j & \text{if } i = m \\ |I_i| - \sum_{T_j \in I_i} C_j + min(sc(I_{i+1}),0) & \text{sinon} \end{cases} \tag{8.3}$$

Calculons à présent les spare capacities des quatre intervalles d'exécution de l'ordonnancement

σ de la figure 8.1 page 192. On a :

$$\begin{cases} sc(I_4) = |I_4| - \sum_{j=5}^{5} C_j = 4-6 = -2 \\ sc(I_3) = |I_3| - \sum_{j=4}^{4} C_j + min(sc(I_4),0) = 6-2-2 = 2 \\ sc(I_2) = |I_2| - \sum_{j=2}^{3} C_j + min(sc(I_3),0) = 4-5+0 = -1 \\ sc(I_1) = |I_1| - \sum_{j=1}^{1} C_j + min(sc(I_2),0) = 6-2-1 = 3 \end{cases} \tag{8.4}$$

On remarque qu'on obtient des spare capacities négatives pour les intervalles d'exécution I_2 et I_4. Dans I_4, par exemple, cela est dû au fait que la tâche T_5 qui appartient à I_4 est de durée supérieure à la taille de cet intervalle d'exécution. Bien que les spare capacities sont censées représenter la quantité de temps disponible sur un intervalle d'exécution et, à ce titre être positives ou nulles, des valeurs négatives permettent de connaître la quantité de temps qu'un intervalle d'exécution "vole" aux intervalles d'exécution précédents. Ainsi nous verrons que garder ces valeurs négatives permet, en-ligne, de mettre à jour les spare capacities par incrément et décrément plutôt que par recalcul de la totalité des spare capacities avec la formule 8.3 page 192.

Cependant, il faut s'en souvenir lors du calcul de la quantité de temps disponible sur une séquence d'intervalles d'exécution, et ne pas se contenter de faire la somme des spare capacities, mais ne considérer que les spare capacities positives. Si on reprend les valeurs de l'exemple de la figure 8.1 page 192, la quantité de temps disponible sur l'intervalle temporel [0,20] est égale à $(sc(I_1) + sc(I_3)) = 5$ ce qui correspond aux temps d'inactivité processeur sur les intervalles temporels [7,10] et [18,20] que l'on peut observer sur la figure 8.1 page 192.

8.2.3 Calculs en-ligne pour l'acceptation de tâches apériodiques

En-ligne, lors de l'exécution, l'ordonnanceur est appelé à la fin de chaque slot afin de définir quelle sera la tâche qui occupera le processeur durant le prochain slot. Cet ordonnanceur commence par répertorier les tâches apériodiques qui ont été activées durant ce slot. Puis, s'il y a de nouvelles tâches apériodiques fermes, un test d'acceptation est appliqué à chacune d'elles. Ensuite, l'ordonnanceur choisit la tâche, périodique, apériodique ferme ou souple, qui sera exécutée dans le prochain slot. Enfin, en fonction de cette dernière décision, les spare capacities sont mises à jour.

8.2.3.1 Prise en compte des tâches apériodiques

La première étape consiste donc à répertorier les tâches apériodiques qui ont été activées lors du slot courant.

En ce qui concerne les tâches apériodiques souples, puisqu'elles n'ont pas d'échéance, l'ordre dans lequel elles seront traitées n'est pas important. Chaque nouvelle tâche apériodique souple est

donc ajoutée en queue de la liste des tâches apériodiques souples ($\mathcal{L}_{ap.s.}$) préalablement activées mais non encore entièrement exécutées.

En ce qui concerne les tâches apériodiques fermes, elles sont ajoutées dans une liste des tâches dont on doit juger si elles peuvent être acceptées ($\mathcal{L}_{acceptation}$). L'ordre n'a pas d'importance.

8.2.3.2 Test d'acceptation

Les étapes suivantes sont répétées jusqu'à ce que la liste $\mathcal{L}_{acceptation}$ soit vide.

Chaque tâche apériodique ferme de la liste $\mathcal{L}_{acceptation}$ possède une échéance et une durée d'exécution. Une telle tâche est notée $J_A(r_A,C_A,d_A)$. Pour savoir si à la date $t = r_A$ on peut accepter cette tâche, c'est-à-dire être sûr que son échéance sera respectée, on se sert des spare capacities pour connaître la quantité de temps disponible sur l'intervalle temporel $]t,d_A]$. Cette quantité se décompose en trois parties :

- $sc(I_c)$, qui est la spare capacity restante de l'intervalle d'exécution I_c tel que $t \in I_c$,
- $\sum sc(I_i)$, où $c < i \leq l$, $end(I_l) \leq d_A < end(I_{l+1})$ et $sc(I_i) > 0$, la somme des spare capacities positives des intervalles d'exécution entièrement inclus dans $]t,d_A]$,
- $\max(\min(sc(I_{l+1}),d_A - start(I_{l+1})),0)$, la spare capacity, si elle est positive, de l'intervalle d'exécution contenant d_A ($d_A \in I_{l+1}$) mais en ne comptant au maximum que la durée séparant $start(I_{l+1})$ de d_A.

Si la somme de ces trois parties est supérieure ou égale à C_A, alors la tâche peut être acceptée. Cela ne veut pas dire qu'elle va être exécutée dans le prochain slot, mais seulement que son échéance sera satisfaite au même titre que l'ensemble des contraintes des tâches périodiques et des tâches apériodiques fermes préalablement acceptées. La tâche acceptée est retirée de la liste $\mathcal{L}_{acceptation}$ et ajoutée à la liste $\mathcal{L}_{ap.f.}$ des tâches apériodiques fermes préalablement acceptées, où les tâches sont ordonnées par échéances croissantes. Si au contraire la somme des trois parties est inférieure à C_A, la tâche est rejetée définitivement et enlevée de la liste $\mathcal{L}_{acceptation}$. La complexité de ce test d'acceptation est en $O(I)$ où I est le nombre d'intervalles d'exécution séparant t de d_A.

Si la tâche apériodique ferme est acceptée, les spare capacities doivent être mises à jour pour prendre en compte le fait qu'une partie du temps disponible est dorénavant réservée à J_A et ne peut donc servir à l'acceptation d'une nouvelle tâche. Si d_A n'est pas égale à l'une des échéances de l'ordonnancement hors-ligne (les bornes des intervalles d'exécution), un nouvel intervalle d'exécution doit être créé. L'intervalle d'exécution I_k tel que $d_A \in I_k$ est donc remplacé par deux intervalles d'exécution[2] I'_k :$]start(I_k),d_A]$ et I''_k :$]d_A,end(I_k)]$. La tâche apériodique ferme J_A est la seule appartenant à I'_k, tandis que toutes les tâches qui appartenaient à I_k appartiennent maintenant à I''_k. Dans le cas où il existe un intervalle d'exécution I_j tel que $end(I_j) = d_A$, la tâche apériodique ferme est simplement rajoutée à la liste des tâches appartenant à cet intervalle d'exécution.

Puisqu'il y a une nouvelle tâche, les spare capacities doivent être mises à jour. Dans le cas où un intervalle d'exécution I_k a été remplacé par deux intervalles d'exécution I'_k (auquel appartient la tâche apériodique acceptée J_A) et I''_k (auquel appartiennent toutes les tâches qui appartenaient à I_k), on recalcule tout d'abord la spare capacity de l'intervalle d'exécution I''_k. Cette spare capacity

2. Dans tous les cas, il est important que les indices des intervalles d'exécution restent croissant. On pourra par exemple utiliser l'indice du slot, relativement à l'hyperpériode, qui correspond au début de l'intervalle d'exécution.

peut être obtenue grâce à celle de l'intervalle d'exécution qui vient d'être remplacé (I_k), en lui soustrayant la taille $|I'_k|$ de l'intervalle d'exécution I'_k. En effet, par rapport à l'intervalle d'exécution initial I_k, l'intervalle d'exécution I''_k possède les mêmes tâches et précède un intervalle d'exécution dont la spare capacity reste inchangée, mais sa taille est réduite de $|I'_k|$ par rapport à $|I_k|$: c'est la seule chose qui a changé si on utilise la formule 8.3 page 192 qui donne la spare capacity d'un intervalle d'exécution. Ensuite, il convient de calculer la spare capacity de l'intervalle d'exécution I'_k auquel appartient J_A (en utilisant la même formule 8.3 page 192) puis, s'il y a lieu, les spare capacities des intervalles d'exécution précédents jusqu'à l'intervalle d'exécution courant.

Alors que les articles de Fohler indiquent qu'on doit mettre à jour les spare capacities des intervalles d'exécution de droite à gauche, en partant de l'intervalle d'exécution auquel appartient J_A jusqu'à l'intervalle d'exécution courant, nous donnons ici une condition permettant l'arrêt des mises à jour avant d'atteindre l'intervalle d'exécution courant.

THÉORÈME 8.1
Soit une suite d'intervalles d'exécution et les spare capacities associées. Soit I_c l'intervalle d'exécution courant et I_f l'intervalle d'exécution auquel appartient la tâche apériodique nouvellement acceptée. Si $sc(I_i)$ désigne la nouvelle spare capacity d'un intervalle d'exécution I_i et $sc_{old}(I_i)$ l'ancienne, les spare capacities sont recalculées en partant de I_f et en remontant l'ordonnancement de droite à gauche. Il est possible de s'arrêter après avoir recalculé la spare capacity de l'intervalle d'exécution I_i si et seulement si :

$$(sc_{old}(I_i) = sc(I_i))\, or\, ((sc(I_i) \geq 0)\, and\, (sc_{old}(I_i) \geq 0)) = TRUE \tag{8.5}$$

DÉMONSTRATION DU THÉORÈME 8.1
Preuve du SI : *En dehors du dernier intervalle d'exécution de l'hyperpériode, la spare capacity de l'intervalle d'exécution précédent I_i, notée I_{i-1}, est obtenue par la formule $sc(I_{i-1}) = |I_{i-1}| - \sum_{T_j \in I_{i-1}} C_j + min(sc(I_i),0)$. Or ni la taille, ni le nombre et la durée des tâches appartenant à I_i n'ont changé. Si il y a changement de la spare capacity, il provient du changement de la valeur de $min(sc(I_i),0)$. Or si $sc_{old}(I_i) = sc(I_i)$, on a $min(sc(I_i),0) = min(sc_{old}(I_i),0)$ et donc $sc_{old}(I_{i-1}) = sc(I_{i-1})$. De même si $((sc(I_i) \geq 0)\, and\, (sc_{old}(I_i) \geq 0)$, on a $min(sc(I_i),0) = min(sc_{old}(I_i),0) = 0$ et donc $sc_{old}(I_{i-1}) = sc(I_{i-1})$. Le fait qu'une spare capacity reste inchangée étant inclus dans la condition 8.5 page 195, on a en outre $sc_{old}(I_i) = sc(I_i) \Rightarrow sc_{old}(I_{i-1}) = sc(I_{i-1})$. Ce qui montre que si la condition 8.5 page 195 est remplie, alors aucun intervalle d'exécution précédent celui-ci ne verrait sa spare capacity modifiée par un recalcul utilisant la formule 8.3 page 192. Il est donc possible d'arrêter les mises à jour après cet intervalle d'exécution* ⊡
Preuve du SEULEMENT SI : *Raisonnement similaire, le terme $min(sc(I_i),0)$ ne pouvant varier que si $sc_{old}(I_i) \neq sc(I_i)$ et que $((sc(I_i) < 0)$ ou $(sc_{old}(I_i) < 0)$* ⊡

Il est à noter que la durée d'exécution de la tâche apériodique ferme est prise en compte dans l'intervalle d'exécution auquel elle appartient, celui correspondant à son échéance. Ainsi on considère le cas où son exécution se fera le plus tard possible ce qui revient à utiliser l'algorithme d'ordonnancement Earliest Deadline Late (EDL). Comme nous le montre le théorème 6.12 page 164 de Chetto et Chetto, cela optimise le temps disponible pour les exécutions de tâches apériodiques.

8.2.3.3 Ordonnancement en-ligne

Une fois les tâches apériodiques fermes acceptées ou rejetées, il reste à décider, à une date t quelle tâche va être exécutée dans le prochain slot. Parmi les tâches périodiques appartenant à un ordonnancement hors-ligne σ, à chaque instant il y a au plus une tâche qui puisse être exécutée. L'ordonnancement est répété indéfiniment, ce qui revient à compter le temps modulo l'hyperpériode. A toute date t, une fonction permet de retourner T_k telle que :

- T_k n'a pas terminé son exécution dans cette hyperpériode,
- $\forall i, i < k$, T_i a terminé son exécution dans cette hyperpériode.

Il est possible que t soit inférieur à la date de début au plus tôt est_k de la tâche T_k : cette tâche ne peut alors pas être exécutée tout de suite. Nous avons aussi la liste $\mathcal{L}_{ap.f.}$ des tâches apériodiques fermes acceptées mais pas encore exécutées, ordonnées par date d'échéance croissantes. Enfin, il y a la liste des tâches apériodiques souples pas encore entièrement exécutées, $\mathcal{L}_{ap.s.}$.

La décision se fait en fonction de ces listes et de la spare capacity de l'intervalle d'exécution courant (I_c tel que $start(I_c) < t \leq end(I_c)$). Si la spare capacity de cet intervalle d'exécution est supérieure à zéro et que $\mathcal{L}_{ap.s.} \neq \varnothing$, la première tâche de $\mathcal{L}_{ap.s.}$ est exécutée dans le prochain slot. Sa durée d'exécution est décrémentée de 1 et si elle devient nulle, la tâche est retirée de la liste (la tâche sera terminée à la fin du prochain slot).

Si la spare capacity de l'intervalle d'exécution courant est nulle [3] ou que $\mathcal{L}_{ap.s.} = \varnothing$, on choisit, entre T_k (si $est_k \leq t$) et la première tâche de la liste $\mathcal{L}_{ap.f.}$, la tâche dont l'échéance est la plus proche [4]. (Earliest Deadline First). En effet, une spare capacity nulle indique que toute décision autre que l'exécution d'une tâche apériodique ferme ou d'une tâche ordonnancée hors-ligne entraînerait un dépassement d'échéance. Là encore, la durée d'exécution de la tâche choisie est décrémentée de 1. Si la tâche appartenait à $\mathcal{L}_{ap.f.}$ et que sa durée d'exécution devient nulle, elle est retirée de cette liste.

Si il n'existe pas de tâche ordonnancée hors-ligne pouvant être exécutée et que $\mathcal{L}_{ap.s.}$ et $\mathcal{L}_{ap.f.}$ sont vides, le processeur reste inoccupé pendant le prochain slot.

8.2.3.4 Mises à jour après ordonnancement en-ligne

En fonction de la décision prise sur l'utilisation du processeur pour le prochain slot, il faut de nouveau mettre à jour les spare capacities. Si cette décision consiste à laisser le processeur inactif durant le prochain slot, il faut décrémenter la spare capacity de l'intervalle d'exécution courant de 1. Si la décision consiste à exécuter une tâche apériodique souple durant le prochain slot, il faut là encore décrémenter la spare capacity de l'intervalle d'exécution courant de 1. Par contre s'il s'agit d'une tâche apériodique ferme ou d'une tâche périodique ordonnancée hors-ligne, cela dépend de l'intervalle d'exécution auquel elle appartient. Si elle appartient à l'intervalle d'exécution courant, son exécution est déjà prise en compte dans le calcul de sa spare capacity et aucune mise à jour n'est nécessaire. Si par contre, elle appartient à un autre intervalle d'exécution, il faut modifier les spare capacities de l'intervalle d'exécution courant et de celui auquel appartient la tâche choisie

3. la spare capacity de l'intervalle d'exécution courant ne peut pas être négative, nous le verrons dans 8.2.3.4 page 196

4. en cas d'égalité on privilégiera la tâche apériodique ferme

de manière à ce qu'elles prennent en compte le fait que cette tâche s'exécute en dehors de son intervalle d'exécution. Les spare capacities des intervalles d'exécution compris entre l'intervalle d'exécution courant et celui auquel appartient la tâche choisie peuvent également être modifiées. Cette mise à jour se déroule alors en trois étapes :

- **étape 1 :** la spare capacity de l'intervalle d'exécution courant I_c est décrémentée de 1 ;
- **étape 2 :** la spare capacity de l'intervalle d'exécution I_x auquel appartient la tâche est incrémentée de 1 ;
- **étape 3 :** si l'ancienne valeur de $sc(I_x)$ était négative, la spare capacity de l'intervalle d'exécution le précédant est elle aussi incrémentée de 1. Tant que l'ancienne valeur de la dernière spare capacity incrémentée était négative, on incrémente celle de l'intervalle d'exécution précédent. La spare capacity de l'intervalle d'exécution courant ne pouvant jamais être négative[5], cette mise à jour s'arrête au plus tard à cet intervalle d'exécution. La propagation des ces incrémentations s'explique par le fait qu'un intervalle d'exécution dont la spare capacity était négative "vole", une fois celle-ci incrémentée, un slot de moins qu'auparavant à l'intervalle d'exécution le précédant.

La modification des échéances comme expliquée dans la section 7.4 permet de limiter le nombre d'intervalles d'exécution et les cas où une tâche s'exécute en dehors du sien. Elle permet ainsi de réduire les calculs en-ligne.

8.2.3.5 Exemple

Revenons à l'exemple 8.1 page 192 avec ses quatre intervalles d'exécution et les spare capacities correspondantes :

- $I_1 :]0,6]$, $sc(I_1) = 3$,
- $I_2 :]6,10]$, $sc(I_2) = -1$,
- $I_3 :]10,16]$, $sc(I_3) = 2$,
- $I_4 :]16,20]$, $sc(I_4) = -2$.

A la date $t = 0$, en absence de tâches apériodiques, c'est la tâche T_1 qui est choisie pour être exécutée durant le premier slot. Puisqu'elle appartient à l'intervalle d'exécution courant, il n'y a pas de mise à jour à effectuer.

Durant le premier slot, la tâche apériodique ferme $J_1(1,2,7)$ est activée. Elle est prise en compte à la fin de ce slot ($t = 1$), où on teste son acceptation. Comme la spare capacity de l'intervalle d'exécution courant est supérieure à C_{J_1}, la tâche est acceptée. Son échéance définit un nouvel intervalle d'exécution $I_{J_1} :]6,7]$ qui raccourcit l'intervalle d'exécution I_2. La figure 8.2 page 198 montre, à la fin du premier slot, les intervalles d'exécution ainsi que l'ordonnancement attendu (si aucune autre tâche apériodique n'était ensuite activée).

On recalcule les spare capacities. $sc(I_4)$ et $sc(I_3)$ restent inchangées. On obtient $sc(I_2)$ en soustrayant à l'ancienne valeur la taille de l'intervalle d'exécution qui vient d'être crée ($|I_{J_1}|$). Puis

5. le cas échéant cela impliquerait que la durée cumulée des tâches restant à exécuter avant la prochaine échéance est supérieure au temps disponible d'ici là.

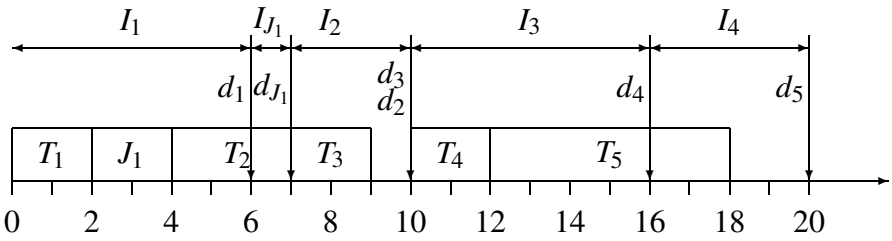

FIG. 8.2 – *Ordonnancement prévu à la fin du premier slot, après acceptation de la tâche apériodique ferme* J_1

on utilise l'équation 8.3 page 192 pour mettre à jour $sc(I_{J_1})$ et $sc(I_1)$. Les spare capacities des intervalles d'exécution sont alors les suivantes :

- $sc(I_1) = 1$,
- $sc(I_1) = -3$,
- $sc(I_2) = -2$,
- $sc(I_3) = 2$,
- $sc(I_4) = -2$.

Ensuite, on choisit la tâche qui sera exécutée dans le deuxième slot. La tâche périodique T_1 poursuit son exécution car elle a son échéance plus tôt que J_1. Il n'y a pas de mise à jour des spare capacities à effectuer en fonction de ce choix car T_1 appartient à l'intervalle d'exécution courant, I_1.

A la date $t = 2$, c'est la tâche apériodique ferme J_1 qui est choisie pour être exécutée dans le troisième slot. Comme elle appartient à l'intervalle d'exécution I_{J_1} et non pas à l'intervalle d'exécution courant I_1, la spare capacity de l'intervalle d'exécution courant est décrémentée et celle de l'intervalle d'exécution I_{J_1} est incrémenté. Comme celle-ci était négative, on incrémente également celle de l'intervalle d'exécution précédent I_{J_1}. Décrémentée puis incrémentée, la spare capacity de I_1 reste au final inchangée. La seule spare capacity ayant changé est $sc(I_{J_1}) = -2$.

A la date $t = 3$, le raisonnement est le même, la tâche J_1 est choisie pour achever son exécution. La spare capacity de I_{J_1}, l'intervalle d'exécution auquel appartient J_1, est décrémentée : $sc(I_{J_1}) = -1$. La spare capacity de l'intervalle d'exécution courant, décrémentée puis incrémentée, reste inchangée.

A la date $t = 4$, on teste l'acceptation de la tâche apériodique ferme $J_2(4,3,11)$, activée durant le quatrième slot. Le temps disponible sur $]4,11]$ est composé de la spare capacity de I_1 qui est égale à 1 et de $\max(\min(sc(I_3), d_{J_2} - start(I_3)), 0) = 1$, les spare capacities des intervalles d'exécution I_{J_1} et I_2 étant négatives. On arrive donc à un total de 2 slots, ce qui est insuffisant. La tâche J_2 est donc rejetée. La tâche T_2 est ensuite choisie pour s'exécuter dans le cinquième slot. Comme elle appartient à I_2 et non pas à l'intervalle d'exécution courant, la spare capacity de celui-ci, I_1, est décrémentée tandis que celle de l'intervalle d'exécution I_2 est incrémentée. Comme elle était négative, il faut aussi incrémenter la spare capacity de l'intervalle d'exécution précédent, I_{J_1} et la condition d'arrêt 8.5 page 195 n'étant pas remplie, on incrémente aussi I_1. On a donc $sc(I_1) = 1$, $sc(I_{J_1}) = 0$ et $sc(I_2) = -1$.

A la date $t = 5$, la tâche T_2 est choisie pour poursuivre son exécution, la spare capacity de I_1 est décrémentée, celle de I_2 incrémentée, entraînant l'incrémentation de celle de J_1. On a donc $sc(I_1) = 0$, $sc(I_{J_1}) = 1$ et $sc(I_2) = 0$.

A la date $t = 6$, la tâche T_2 est choisie pour achever son exécution. L'intervalle d'exécution courant est I_{J_1}, dont la spare capacity est décrémentée alors que celle de I_2 incrémentée. On a donc $sc(I_{J_1}) = 0$ et $sc(I_2) = 1$.

A la date $t = 7$, on teste l'acceptation de la tâche apériodique ferme $J_3(7,2,16)$, activée durant le septième slot. Le temps disponible sur $]7,16]$ est composé de la spare capacity de I_2 qui est égale à 1 et de la spare capacity de l'intervalle d'exécution I_3 qui est égale à 2. La tâche est donc acceptée. Comme $d_{J_3} = end(I_3)$, il n'y a pas d'intervalle d'exécution à rajouter. On a juste à soustraire C_{J_3} à la spare capacity de I_3. Celle si devient nulle et n'étant pas négative ne nécessite pas de mise à jour des autres spare capacity. La figure 8.3 page 199 montre les intervalles d'exécution ainsi que l'ordonnancement attendu (si aucune autre tâche apériodique n'était ensuite activée).

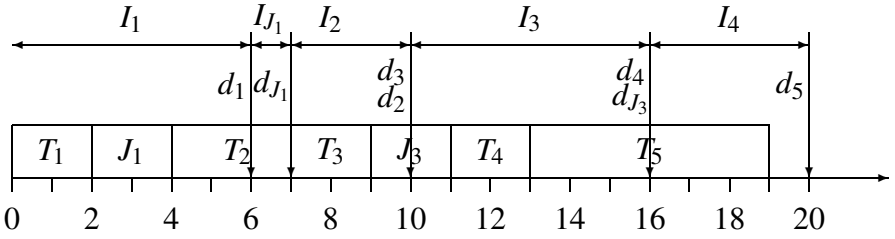

FIG. 8.3 – *Ordonnancement prévu à la fin du septième slot, après acceptation de la tâche apériodique ferme J_3*

On a alors les spare capacities suivantes :

- $sc(I_1) = 0$,
- $sc(I_1) = 0$,
- $sc(I_2) = 1$,
- $sc(I_3) = 0$,
- $sc(I_4) = -2$.

La tâche choisie pour être exécutée est T_3 car son échéance est plus proche que celle de J_3. Comme elle appartient à l'intervalle d'exécution courant, il n'y a pas de mise à jour à effectuer.

A la date $t = 8$, la liste des tâches apériodiques souples $\mathcal{L}_{ap.s.}$ n'est pas vide, une tâche, $J_4(8,1)$, ayant été activée durant le huitième slot. Comme la spare capacity de l'intervalle d'exécution courant n'est pas nulle, elle est exécutée dans le slot suivant. La spare capacity de l'intervalle d'exécution courant est décrémentée de 1. Il ne reste plus de spare capacities strictement positives sur l'hyperpériode, les tâches seront donc exécutées dans l'ordre donné par la figure 8.4 page 200. En effet il ne peut y avoir d'exécution de tâches supplémentaires avant $t = 20$.

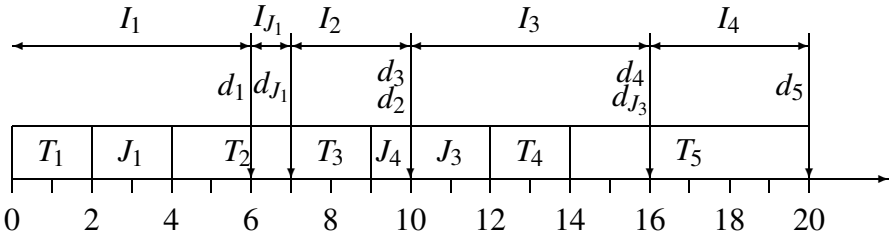

FIG. 8.4 – *Ordonnancement observé sur l'hyperpériode, avec exécution des tâches apériodiques fermes acceptées J_1 et J_3 ainsi que de la tâche apériodique souple J_4*

8.2.4 Optimalité

Le slot shifting mêle acceptation (et donc exécution) de tâches apériodiques fermes et exécution de tâches apériodiques souples. Il faut donc, pour discuter de l'optimalité, se restreindre successivement à l'acceptation de tâches apériodiques fermes puis à l'exécution de tâches apériodiques souples.

Dans [2], Fohler a démontré que le test d'acceptation des tâches apériodiques fermes du slot shifting combiné à la gestion des spare capacities était optimal car équivalent aux conditions d'ordonnançabilité des articles [143], [144] et [122]. L'optimalité* atteignable en matière d'acceptation de tâches apériodiques fermes est donnée par la définition suivante : *soit un ordonnancement hors-ligne σ de tâches périodiques avec date de début au plus tôt et des échéances et une liste $\mathcal{L}$ de tâches apériodiques fermes ordonnées par date d'activation. Un test d'acceptation permet d'accepter ou rejeter une tâche apériodique ferme : dans le cas où elle est acceptée celle-ci est assurée de respecter son échéance sans que cela nuise au respect des échéances des autres tâches périodiques ou apériodiques, préalablement acceptées. Un test d'acceptation est optimal si la première tâche apériodique de la liste $\mathcal{L}$ rejetée par tout autre test d'acceptation est égale à celle rejetée par ce test ou la précède dans la liste $\mathcal{L}$.*

Dans les travaux de Fohler, la question de l'optimalité des temps de réponse des tâches apériodiques souples n'est, à notre connaissance, pas abordée. Pourtant, en absence de tâches apériodiques fermes, il est possible de prouver que le slot shifting est optimal car il minimise les temps de réponse des tâches apériodiques souples. C'est ce que prouve la démonstration du théorème 8.2 que nous proposons.

THÉORÈME 8.2
Soit un ordonnancement hors-ligne σ de tâches périodiques avec date de début au plus tôt et des échéances qui sont ensuite ordonnancées avec EDF. En absence de tâches apériodiques fermes, parmi les méthodes prenant en compte les tâches apériodiques dans leur ordre d'arrivée (FIFO), le slot shifting permet de minimiser le temps de réponse de chaque tâche apériodique souple tout en respectant les échéances des tâches périodiques.

DÉMONSTRATION DU THÉORÈME 8.2
Preuve par contradiction. Soit une tâche apériodique souple J_a. Soit σ' l'ordonnancement obtenu par le slot shifting et mêlant la tâche J_a aux tâches périodiques de l'ordonnancement hors-ligne. Si

le théorème est faux, il existe un ordonnancement σ'' dans lequel J_a termine son exécution plus tôt que dans σ'. Avec le slot shifting une tâche apériodique souple s'exécute dès que la spare capacity de l'intervalle d'exécution courant n'est pas nulle. Si J_a termine son exécution plus tôt dans σ'' que dans σ', c'est qu'il existe une date t dans σ'' où la spare capacity de l'intervalle d'exécution courant est nulle mais où J_a s'exécute quand même. Or si une spare capacity nulle permettait d'exécuter immédiatement une tâche, le slot shifting ne pourrait être optimal pour les tâches apériodiques fermes puisque l'acceptation de tâches est cautionnée par l'existence de spare capacities positives. Il y a donc contradiction, ce qui prouve le théorème □

Le slot shifting atteint donc la même optimalité pour l'exécution de tâches apériodiques souples que la version optimale du serveur à utilisation totale (voir 6.3.2.8 page 165).

8.3 Contraintes de latence et slot shifting

Le slot shifting repose sur la notion d'intervalle d'exécution et des spare capacities associées. Afin d'utiliser le slot shifting pour l'acceptation de tâches apériodiques fermes et souples tout en garantissant les contraintes de latence, il convient de définir comment ces contraintes de latence influent sur les intervalles d'exécution.

Nous rappelons que toutes les données temporelles sont exprimées en nombre de slots et que les tâches possèdent des échéances qui ont été uniformisées comme expliqué dans la sous-section 7.4.2 page 183. Elles possèdent également des contraintes de latence [6]. La figure 8.5 page 201 montre un exemple d'ordonnancement monoprocesseur comportant 4 tâches possédant la même échéance $d = 10$. Il existe en outre une contrainte de latence liant T_2 et T_4 telle que $L(T_2,T_4) = 7$. Les caractéristiques de l'ordonnancement sont données dans le tableau 8.1.

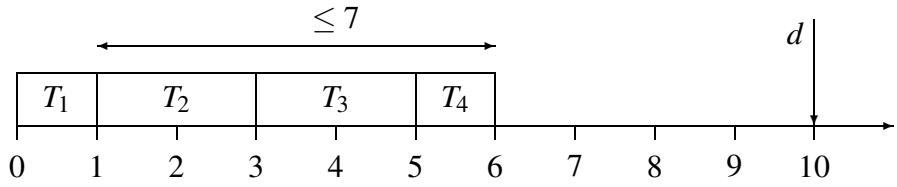

FIG. 8.5 – *Exemple d'ordonnancement possédant une échéance et une contrainte de latence*

Si on considère que l'unique échéance crée un intervalle d'exécution unique, la spare capacity correspondante a pour valeur 4. Cette spare capacity permettrait à une tâche apériodique ferme $J_1(2,3,6)$ activée après l'exécution du premier slot de T_2, d'être acceptée car la spare capacity de l'intervalle d'exécution est supérieure au nombre de slots demandés. Pourtant, l'exécution de cette tâche apériodique ferme avant son échéance, porte à 8 le nombre de slots séparant le début de T_2 de la fin de T_4 : la contrainte de latence $L(T_2,T_4)$ n'est plus respectée.

6. L'ensemble des contraintes de latence a été préalablement réduit en utilisant les règles décrites dans la sous-section 7.4.2 page 183.

T_i	est_i	C_i	d_i	$L(T_h,T_i)$
T_1	0	1	10	
T_2	1	2	10	
T_3	3	2	10	
T_4	5	1	10	$L(T_2,T_4)=7$

TAB. 8.1 – *Caractéristiques de l'ordonnancement de la figure 8.5 page 201*

À partir du moment où T_2 débute son exécution, la latence devient réellement une échéance pour T_4 ainsi que les tâches la précédant (par modification des échéances). Étant donné l'ordonnancement initial hors-ligne, le début d'exécution de T_2 à la date $t=1$ entraîne une échéance notée d_1 à $t = est_2 + L_{24} = 8$, comme l'indique la figure 8.6 page 202. On obtient donc deux intervalles d'exécution $I_1]0,8]$ et $I_2]8,10]$.

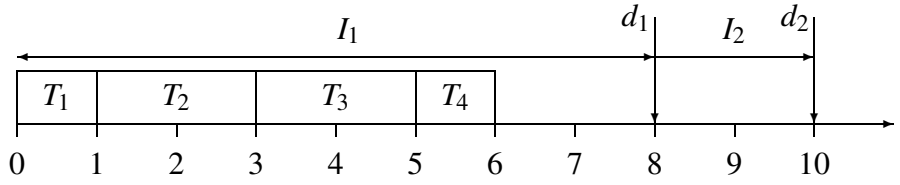

FIG. 8.6 – *Expression de la latence comme une échéance pour permettre le slot shifting*

En ce qui concerne l'appartenance des tâches à ces deux intervalles d'exécution, on peut d'abord appliquer la règle usuelle de modification des échéances. Si la tâche T_4 possède une échéance plus proche que celle des tâches la précédant, on peut remplacer l'échéance de ces dernières par l'échéance de T_4. Les tâches T_1, T_2, T_3 et T_4 possèdent alors toutes la même échéance et appartiennent toutes à I_1, ce qui donne les spare capacities suivantes :

$$\begin{cases} sc(I_2) = |I_2| = 2 \\ sc(I_1) = |I_1| - \sum_{T_i \in I_1} C_i + \min(sc(I_2),0) = 8-(1+2+2+1)+0 = 2 \end{cases} \tag{8.6}$$

On obtient bien un total de 4 slots disponibles sur $]0,10]$. Prenons l'exemple d'une tâche apériodique ferme $J_2(1,4,5)$. Durant le premier slot, la tâche T_1 appartenant à I_1 a été exécutée et donc les spare capacities sont restées inchangées. La quantité de spare capacity disponible entre l'activation de la tâche J_2 et son échéance, c'est-à-dire sur l'intervalle temporel $]1,5]$, est égale à $sc(I_1) = 2$. Ceci est insuffisant pour exécuter J_2 qui est donc rejetée. Or, cette tâche apériodique ferme peut être exécutée avant son échéance et cela en respectant les échéances et la contrainte de latence des tâches périodiques, comme le montre l'ordonnancement de la figure 8.7 page 203.

Cela est dû au fait que l'exécution de J_2 sur l'intervalle temporel $]1,5]$ repousse [7] la date de début d'exécution de T_2 ce qui a pour effet de décaler d'autant l'échéance qu'implique la contrainte de latence sur T_4. On a alors $d_1 = 5+7 = 12$ qui devient ainsi moins contraignante que $d_2 = 10$.

7. En effet, J_2 possède une échéance plus proche que celle de T_2. Elle est donc choisie pour être exécutée la première.

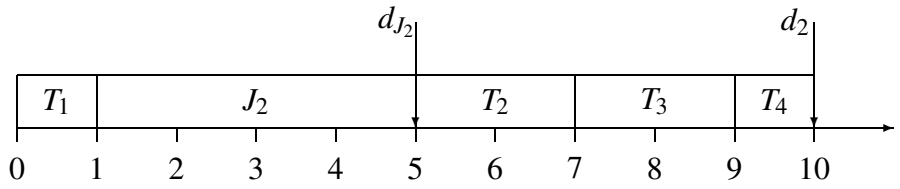

FIG. 8.7 – *Ordonnancement où $J_2(1,4,5)$ est acceptée et où les échéances et la contrainte de latence sont satisfaites*

Puisque les spare capacities précédentes ne permettent pas d'accepter une tâche apériodique ferme acceptable, il faut trouver une autre solution. Si on n'utilise pas la règle de modification des échéances, la tâche T_1 peut garder son échéance $d_2 = 10$ et appartenir à l'intervalle d'exécution I_2. En effet T_1 s'exécutant avant T_2, elle n'est pas réellement concernée par l'échéance d_1 issue de la latence $L(T_2,T_4)$. On obtient alors les spare capacities suivantes :

$$\begin{cases} sc(I_2) = |I_2| - C_1 = 2 - 1 = 1 \\ sc(I_1) = |I_1| - \sum_{T_i \in I_1} C_i + \min(sc(I_2),0) = 8 - (1+3+1) + 0 = 3 \end{cases} \tag{8.7}$$

Mais l'exécution de T_1 dans le premier slot implique une incrémentation de la spare capacity de son intervalle d'exécution I_2 et une décrémentation de celle de l'intervalle d'exécution courant I_1, ce qui donne $sc(I_2) = 2$ et $sc(I_1) = 2$, ce qui ne permet pas non plus d'accepter $J_2(1,4,5)$.

Une dernière solution serait de considérer que l'échéance d_1 ne concerne pas le premier slot de T_2. En effet, ce n'est que la première fois que T_2 est choisie pour être exécutée dans le slot suivant que cette échéance devient réellement un échéance pour les tâches T_4, T_3 et T_2. On peut donc considérer que l'échéance du premier slot de T_2 est d_2 alors que l'échéance du slot suivant de cette même tâche est d_1. Une même tâche ne pouvant pas posséder des slots aux échéances différentes, on coupe la tâche T_2 en deux tâches T_{2a} et T_{2b}. On obtient alors l'ordonnancement de la figure 8.8 page 203.

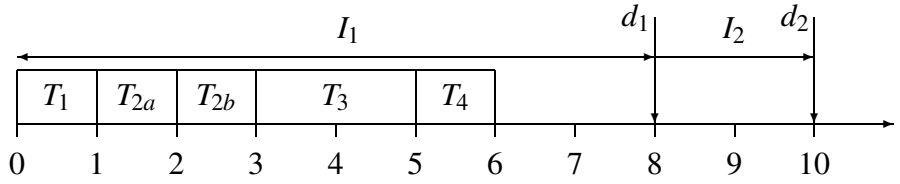

FIG. 8.8 – *Découpage de la tâche T_2 en deux tâches T_{2a} et T_{2b} appartenant à des intervalles d'exécution différents*

On a alors T_1 et T_{2a} qui appartiennent à I_2, et T_{2b}, T_3 et T_4 qui appartiennent à I_1. On obtient les spare capacities suivantes :

$$\begin{cases} sc(I_2) = |I_2| - C_1 - C_{2a} = 2 - 2 = 0 \\ sc(I_1) = |I_1| - \sum_{T_i \in I_1} C_i + \min(sc(I_2),0) = 8 - (1+2+1) + 0 = 4 \end{cases} \tag{8.8}$$

Il est à noter que cette configuration est la seule des trois qui permette l'acceptation d'une tâche qui demanderait 4 slots à $t = 0$ avec une échéance dans l'intervalle d'exécution I_1. Mais en absence d'une telle tâche, l'exécution de T_1 dans le premier slot implique une incrémentation de la spare capacity de son intervalle d'exécution I_2 et une décrémentation de celle de l'intervalle d'exécution courant I_1, ce qui donne $sc(I_2) = 1$ et $sc(I_1) = 3$, ce qui ne permet pas non plus d'accepter $J_2(1,4,5)$.

Parmi ces trois configurations d'appartenance des tâches aux intervalles d'exécution, aucune ne permet de traduire correctement les contraintes de latences en slots disponibles. Cela est dû au fait que la contrainte de latence, contrairement à une échéance, est une contrainte relative à une information qui n'est disponible qu'en-ligne (s_i). Si une échéance classique est fixe, une échéance due à une contrainte de latence $L(T_i,T_j)$ peut être mobile car la date de début de la tâche T_i peut varier. Or c'est ce qui arrive lorsque des tâches apériodiques sont exécutées. Cette mobilité des échéances influe sur les intervalles d'exécution et les spare capacities associées. Nous avons également vu que la mobilité d'une échéance due à une contrainte de latence $L(T_i,T_j)$ ne prend fin que lorsque la tâche T_i débute son exécution, ce qui implique que l'on doive considérer séparément le premier slot de tâche T_i et les slots suivants de cette tâche.

L'application du slot shifting aux contraintes de latence exprimées par des échéances n'est donc pas possible sans quelques modifications, parce que ces échéances ne sont pas fixes mais mobiles. Il existe alors deux possibilités distinctes :

- soit on traduit hors-ligne les contraintes de latence en échéances et on introduit dans les calculs en-ligne la gestion de la mobilité de ces échéances,
- soit on traduit en-ligne chaque contrainte de latence $L(T_i,T_j)$ par une échéance, uniquement lorsque celle-ci ne peut plus varier, c'est-à-dire quand la tâche T_i est choisie pour être exécutée dans le prochain slot.

Nous avons exploré dans ce qui suit les deux solutions afin de trouver la mieux adaptée. Nous verrons que la première augmente considérablement la complexité des calculs en-ligne, ce qui n'est pas envisageable dans un contexte temps réel et cela même si le coût de ces calculs peut être inclus dans la durée du slot. La seconde solution, quant à elle, ne nécessite que peu de modifications dans les calculs en-ligne, mais la possibilité de son application à une contrainte de latence donnée dépend des relations de cette contraintes avec les autres contraintes de latence et d'échéance. Nous verrons que dans ce cas les contraintes de latence ne pouvant pas être prises en compte sont traduites hors-ligne en échéances fixes.

8.4 Slot shifting où les latences sont traduites hors-ligne par des échéances mobiles

8.4.1 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques avec contraintes de latence

Avant que les contraintes de latence soient traduites par des échéances, il y a eu uniformisation des échéances de l'ordonnancement hors-ligne. Pour chaque contrainte de latence $L(T_i,T_j)$ on assigne à T_j une échéance mobile d_j telle que $d_j = est_i + L_{ij}$. Si $C_i > 1$ on remplace T_i par

deux tâches T_i et $T_{i'}$ de manière à pouvoir leur assigner deux échéances différentes. On note $C_{i_{old}}$ l'ancienne valeur de C_i et $d_{i_{old}}$ l'abcienne valeur de d_i, on définit les deux tâches comme suit :

- $C_i = 1$, la durée d'exécution de la tâche T_i vaut à présent 1,
- $C_{i'} = C_{i_{old}} - 1$, la tâche $T_{i'}$ possède une durée d'exécution inférieure d'un slot à la durée d'exécution initiale de la tâche T_i,
- $est_{i'} = est_i + 1$,
- $i < i'$, T_i précède $T_{i'}$ dans l'ordonnancement,
- $d_i = d_{i_{old}}$, la tâche T_i garde l'échéance intiale,
- $d_{i'} = est_i + L_{ij}$, l'échéance de $T_{i'}$ tient compte de la contrainte de latence.

Toutes les échéances définissent des intervalles d'exécution sur l'ordonnancement. Il convient ensuite de modifier à nouveau les échéances des tâches pour définir les tâches concernées par ces échéances mobiles. Cette mise à jour se fait en considérant les contraintes de latence par ordre décroissant de leur échéance mobile. Pour chaque contrainte de latence $L(T_i,T_j)$ on modifie les échéances des tâches correspondantes comme suit :

$$\forall k, i < k \leq j \quad d_k = d_j \tag{8.9}$$

Une fois cette modification effectuée sur l'ensemble de l'ordonnancement, l'appartenance des tâches aux différents intervalles d'exécution est directement déduite des ces échéances. On calcule ensuite les spare capacities avec la formule habituelle du slot shifting. Pour l'exemple de la figure 8.8 page 203, on obtient $sc(I_2) = 0$ et $sc(I_1) = 4$.

8.4.2 Calculs en-ligne pour l'acceptation de tâches apériodiques

Nous rappelons que ces calculs en-ligne peuvent être découpés en deux phases. La première consiste à tester l'acceptation des tâches apériodiques fermes et à mettre à jour les spare capacities si certaines sont acceptées. La deuxième permet de choisir la tâche à exécuter dans le prochain slot (ordonnancement en-ligne) et à mettre à jour les spare capacities en conséquence.

8.4.2.1 Test d'acceptation

Dans le slot shifting sans contraintes de latence, les spare capacities permettent de connaître directement la quantité de temps disponible pour l'exécution d'une tâche apériodique. L'acceptation de cette tâche ne modifie pas les échéances des autres et il est donc simple de calculer si la quantité de temps disponible est suffisante.

Mais comme nous l'avons vu dans la section 8.3, l'acceptation d'une tâche apériodique ferme peut modifier les échéances des tâches, entraînant un changement des intervalles d'exécution et des spare capacities associées. Il convient donc d'anticiper l'acceptation de la tâche en modifiant virtuellement, c'est-à-dire de manière non définitive, les intervalles d'exécution et les spare capacities afin d'effectuer le test d'acceptation sur cette nouvelle configuration des intervalles d'exécution. Si la tâche apériodique ferme est acceptée, cette configuration devient réellement celle de l'ordonnancement et un nouvel intervalle d'exécution est défini si besoin pour la tâche acceptée, comme dans

le slot shifting traditionnel. Si la tâche apériodique ferme est rejetée, l'ancienne configuration des intervalles d'exécution est restaurée. Toute la difficulté est donc d'obtenir la nouvelle configuration à partir de l'ancienne.

L'algorithme 6 page 207 que nous proposons pour calculer la nouvelle configuration nécessite l'introduction de quelques notations supplémentaires. Nous avons tout d'abord besoin d'une donnée permettant de connaître la date de début envisagée pour une tâche plutôt que sa date de début au plus tôt. Pour chaque tâche T_i sa date de début envisagée est notée dst_i (pour *delayed start time*). Initialement égale à est_i, sa valeur augmente lorsque la date de début d'exécution est repoussée. Par ailleurs, on désigne par T_d la dernière tâche périodique ayant été totalement exécutée. De plus, on note d_{k_0} l'échéance initiale d'une tâche T_k, c'est-à-dire celle effective avant que les contraintes de latence aient été traduites hors-ligne en échéances [8]. Cette échéance est stockée pour chaque tâche T_k dans un vecteur Δ_k contenant toutes les contraintes d'échéance et dont on note la taille $|\Delta_k|$. Si celles-ci étaient fixes il suffirait de considérer uniquement la plus contraignante, c'est-à-dire celle possédant la plus petite valeur. Puisque les échéances dues aux contraintes de latence sont mobiles, il se peut qu'en-ligne l'échéance la plus contraignante varie au cours du temps. C'est pourquoi le vecteur Δ_k contient également $dst_i + L_{ij}$ pour chaque couple (i,j) tel que $L(T_i,T_j)$ existe et que $i \leq k \leq j$. Si on reprend l'exemple de la tâche T_3 de l'ordonnancement illustré par la figure 8.5 page 201, on a :

$$\Delta_3 = (d_{3_0}, dst_2 + L_{24}) \quad \text{avec } d_{3_0} = 10 \tag{8.10}$$

Pour connaître l'échéance la plus contraignante il suffit de chercher le minimum des valeurs que contient ce vecteur. Dans notre exemple à $t = 0$ il s'agit de $dst_2 + L_{24} = 8$, mais si l'exécution de T_2 est repoussée comme sur la figure 8.7 page 203 ce minimum peut être $d_{3_0} = 10$, On rappelle enfin que $\mathcal{L}_{ap.f.}$ est la liste des tâches apériodiques préalablement acceptées mais pas entièrement exécutées.

L'algorithme 6 détermine tout d'abord l'indice e de la première tâche dont le début d'exécution peut être repoussé par l'acceptation de la tâche apériodique ferme $J_a(r_a,C_a,d_a)$ considérée. Ensuite on calcule de combien de slots le début d'exécution de T_e va être repoussé en retranchant aux C_a slots demandés par J_a la quantité de temps disponible avant la date de début d'exécution envisagée de T_e :

$$G = C_a - (dst_e - t_a - \sum_{d<k<e} C_k - \sum_{J_m \in \mathcal{L}_{ap.f.},\, d_m \leq d_e} C_m) \tag{8.11}$$

Dans cette formule :

- $dst_e - t_a$ représente le temps séparant la date d'activation de J_a de la date de début d'exécution envisagée de T_e,
- $\sum_{d<k<e} C_k$ représente le nombre de slots nécessaires à l'exécution des tâches périodiques précédant T_e et non encore exécutées,
- $\sum_{J_m \in \mathcal{L}_{ap.f.},\, d_m \leq d_e} C_m$ représente le nombre de slot nécessaires à l'exécution des tâches apériodiques fermes préalablement acceptées et non encore exécutées qui ont une échéance inférieur à d_e.

8. après uniformisation des échéances.

Algorithme 6 de construction de la nouvelle configuration

1: $e = \min_{d<i\leq N}(d_i \geq d_a)$
2: $G = C_a - (dst_e - t_a - \sum_{d<k<e} C_k - \sum_{J_m \in \mathcal{L}_{ap.f.}, d_m \leq d_e} C_m)$
3: **tant que** (G>0)AND($e \leq N$) **faire**
4: $dst_e = dst_e + G$ nouvelle date de début envisagée pour T_e
5: **si** $dst_e + C_e > d_e$ **alors**
6: FIN DE L'ALGORITHME : T_e ne respectera pas son échéance si J_a est acceptée
7: **fin si**
8: **pour** chaque tâche T_f telle que $L(t_e,T_f)$ existe, f allant de e à N **faire**
9: **pour** chaque tâche T_k telle que $e \leq k \leq f$ **faire**
10: $d_k = \min_{1\leq p\leq|\Delta_k|}(\Delta_k(p))$
11: **fin pour**
12: **fin pour**
13: e=e+1
14: **si** $e \leq N$ **alors**
15: $G = C_a - (dst_e - t_a - \sum_{d<k<e} C_k - \sum_{J_m \in \mathcal{L}_{ap.f.}, d_m \leq d_e} C_m)$
16: **fin si**
17: **fin tant que**
18: FIN DE L'ALGORITHME : la nouvelle configuration a été calculée

Si la valeur de G obtenue est négative où nulle, cela signifie qu'il existe assez de temps d'inactivité processeur prévus par l'ordonnancement hors-ligne sur l'intervalle temporel $]r_a,dst_e]$ pour permettre l'exécution de J_a ainsi que des tâches apériodiques fermes préalablement acceptées et non encore exécutées qui ont une échéance inférieur à d_e. Dans ce cas le début d'exécution de T_e n'est pas repoussé et l'algorithme prend fin.

Mais si $G > 0$, le début d'exécution de T_e est repoussé de G slots. On rentre alors dans la boucle *tant que*. Dans celle-ci, on met tout d'abord à jour la date de début d'exécution envisagée de T_e en prenant en compte le retard causé par J_a. On teste ensuite si cette nouvelle date de début d'exécution reste compatible avec le respect de son échéance. Ce test n'est pas obligatoire pour le calcul de la nouvelle configuration mais évite de calculer entièrement une configuration où des tâches ordonnancées hors-ligne ne pourraient plus respecter leur échéance.

Ensuite pour chaque tâche T_f telle qu'il existe une contrainte de latence $L(T_e,T_f)$, on modifie les échéances des tâches comprises entre T_e et T_f si la contrainte de latence $L(T_e,T_f)$ reste la contrainte la plus contraignante pour ces tâches (utilisation d'une fonction *min*). On considère ensuite la tâche ordonnancée hors-ligne qui suit T_e en incrémentant e. On calcule pour cette nouvelle tâche la valeur de G. Pour calculer entièrement la nouvelle configuration, on reboucle jusqu'à ce que G soit négatif ou nul, ou que l'ensemble de l'ordonnancement hors-ligne ait été parcouru. À chaque itération, l'algorithme peut être stoppé si un tâche ne satisfait plus son échéance.

L'algorithme 6 permet donc de calculer la nouvelle configuration et de savoir si cette configuration respecte les échéances des tâches périodiques de l'ordonnancement hors-ligne initial. Étant donné que les échéances des tâches périodiques ne peuvent être que repoussées, les échéances des tâches apériodiques fermes qui étaient respectées dans l'ancienne configuration ne peuvent l'être dans la nouvelle.

Le calcul des spare capacities correspondant à la nouvelle configuration des intervalles d'exécution permet ensuite, en utilisant le test d'acceptation classique du slot shifting, d'accepter ou de rejeter la tâche apériodique ferme sans incidence sur les échéances des tâches apériodiques fermes préalablement acceptées.

8.4.2.2 Ordonnancement en-ligne

Dans le slot shifting classique, le choix de la tâche à exécuter dans le prochain slot dépend de la valeur de la spare capacity de l'intervalle d'exécution courant. Si celle-ci est positive et non nulle, une tâche apériodique souple est exécutée si $\mathcal{L}_{ap.s.} \neq \emptyset$. Le cas échéant c'est une tâche apériodique ferme ou périodique qui est choisie.

L'exécution d'une tâche apériodique souple dans le prochain slot peut modifier la configuration des intervalles d'exécution, ce qui n'est pas le cas de l'exécution d'une tâche apériodique ferme ou périodique. Pour pouvoir tester la valeur de la spare capacity de l'intervalle d'exécution courant, il convient donc de calculer la nouvelle configuration obtenue si une tâche apériodique souple est exécutée dans le prochain slot.

On teste donc tout d'abord s'il existe une tâche apériodique souple à exécuter, c'est-à-dire si $\mathcal{L}_{ap.s.} \neq \emptyset$. Si c'est le cas, on utilise l'algorithme 6 page 207 pour calculer la nouvelle configuration en prenant la date de fin du prochain slot pour valeur de d_a, puis on recalcule les spare capacities. Enfin on applique l'ordonnancement en-ligne du slot shifting classique (voir 8.2.3.3 page 196).

La nouvelle configuration n'est conservée que si c'est bien une tâche apériodique souple qui est choisie, sinon on restaure l'ancienne.

Si $\mathcal{L}_{ap.s.} = \emptyset$ on applique l'ordonnancement en-ligne du slot shifting classique, sans nécessité de calculer une nouvelle configuration.

Les mises à jour qui suivent l'ordonnancement en-ligne ne diffèrent pas du slot shifting classique.

8.4.3 Complexité

Notons N le nombre de tâches de l'ordonnancement hors-ligne et L le nombre de contraintes de latence.

En ce qui concerne la complexité des calculs hors-ligne, la prise en compte des latences comme échéance ajoute aux calculs hors-ligne du slot shifting traditionnel une deuxième modification des échéances qui requiert une complexité en $O(NL)$ en considérant le pire cas. Peu fréquent, le pire cas correspondrait à des contraintes de latence telles que pour chaque $L(T_i,T_j)$, T_i est une des premières tâches de l'ordonnancement et T_j une des dernières ce qui impliquerait que la quasi-totalité des N tâches soient mises à jour L fois. Hors-ligne cette complexité reste néanmoins acceptable.

C'est en-ligne que la complexité augmente fortement. L'acceptation d'une tâche apériodique ferme, habituellement en $O(I)$ où I est le nombre d'intervalles d'exécution, nécessite l'exécution de l'algorithme 6 page 207. Or la complexité de celui-ci est $O(N^2)$. En effet la boucle *tant que* peut être exécutée au pire cas autant de fois qu'il y a de tâches dans l'ordonnancement σ (c'est-à-dire N) et à chacune de ces itérations, il peut exister une contrainte de latence $L(T_e,T_f)$ entraînant la modification des échéances d'un nombre x de tâches, x pouvant aller de 1 à N. On peut même atteindre une complexité supérieure en $O(kN^2)$ si on considère qu'à chaque itération k contraintes de latence $L(T_e,T_f)$ existent. Ce raisonnement vaut aussi si on ne considère que les tâches apériodiques souples, puisque l'exécution de ce même algorithme peut être nécessaire avant de choisir la tâche à exécuter dans le prochain slot.

Cette complexité en-ligne n'est bien sûr pas acceptable, ce qui rend le slot shifting où les latences sont traduites hors-ligne par des échéances mobiles non utilisable en pratique.

8.4.4 Optimalité

Les contraintes de latence étant traduites en échéances, la seule modification par rapport au slot shifting classique est la gestion de la mobilité de ces échéances. Or, cette ajout ne peut remettre en cause l'optimalité propre au slot shifting classique. D'une part la traduction des contraintes de latence par des échéances mobiles permet de ne pas sur-contraindre l'ordonnancement. D'autre part, le calcul de la nouvelle configuration avant d'appliquer le test d'acceptation d'une tâche apériodique ferme assure que les spare capacities correspondent bien aux valeurs que prendront les échéances mobiles en cas d'acceptation de la tâche apériodique ferme considérée. Donc l'optimalité du test d'acceptation du slot shifting classique telle que décrite à la sous-section 8.2.4 page 200 est non seulement conservée mais étendue aux contraintes de latence des tâches périodiques.

En ce qui concerne l'optimalité des temps de réponse des tâches apériodiques souples lorsque l'on se restreint à ce type de tâche apériodique, l'optimalité démontrée dans la sous-section 8.2.4

page 200 est également conservée pour la même raison.

On a donc, moyennant l'ajout de calculs hors-ligne et en-ligne, élargi l'optimalité du slot shifting pour l'ordonnancement mixte de tâches périodiques avec échéances et de tâches apériodiques souples et fermes au cas où les tâches périodiques possèdent également des contraintes de latence.

8.5 Slot shifting où les latences sont traduites en ligne par des échéances fixes

Pour ne pas augmenter la complexité des calculs en-ligne, nous avons étudié la possibilité d'ignorer en-ligne une contrainte de latence $L(T_i,T_j)$ jusqu'à la date de début d'exécution s_i de la tâche T_i. Avant cette date aucune échéance ne traduit donc cette contrainte. A la date $t = s_i$, on crée une échéance $d = s_i + L_{ij}$ qui devient l'échéance de toute tâche k telle que $i \leq k \leq j$, à moins qu'elle ne possède déjà une échéance plus proche.

8.5.1 Calculs hors-ligne après ordonnancement hors-ligne des tâches périodiques avec contraintes de latence

Puisque les contraintes de latence ne sont traduites en échéances qu'en-ligne, il n'y a, a priori, pas de calculs supplémentaires hors-ligne. Encore faut-il préalablement prouver qu'une contrainte de latence peut être ignorée en-ligne jusqu'à la date où elle devient fixe, sans que les décisions prises (acceptation de tâches apériodiques fermes, exécution de tâches apériodiques) ne puisse causer le non-respect de cette contrainte. Nous verrons qu'il existe des contraintes de latence pour lesquelles il n'est pas possible de prouver une telle propriété. Celles-ci seront traduites hors-ligne par une échéance fixe qui sera traitée en-ligne comme une échéance classique : on perd alors la flexibilité de la contrainte de latence. Nous allons donc présenter d'abord nos résultats qui permettent de discerner les contraintes de latence pouvant être traduites en-ligne de celles qui ne le peuvent pas. Ensuite nous nous servirons de ces résultats pour construire $\mathcal{L}_{hl}$, l'ensemble des contraintes de latence devant être traduites hors-ligne en échéances fixes.

Si on considère un ordonnancement σ où les échéances ont été uniformisées, on distingue tout d'abord pour une contrainte de latence $L(T_i,T_j)$, le cas où $d_i \neq d_j$ du cas où $d_i = d_j$.

8.5.1.1 Cas $d_i \neq d_j$

On différencie dans ce cas l'exécution de tâches apériodiques souples et l'acceptation de tâches apériodiques fermes. D'où ce premier résultat.

THÉORÈME 8.3
Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ. Si après uniformisation des échéances, on a $d_i \neq d_j$, alors l'exécution selon les règles du slot shifting de tâches apériodiques souples sur n'importe quel intervalle de temps $]t_1,t_2]$ tel que $0 \leq t_1 < t_2 \leq s_i$ ne peut entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU THÉORÈME 8.3
En absence de tâche apériodique, les tâches T_i et T_j sont exécutées aux dates prévues par l'ordonnancement σ et la contrainte de latence est respectée car

$$f_j - s_i = (est_j + C_j) - est_i \leq L_{ij} \tag{8.12}$$

L'exécution dans un slot d'une tâche apériodique souple peut repousser l'exécution des tâches appartenant à l'intervalle d'exécution courant ainsi qu'aux intervalles d'exécution suivants. Les dates s_i et s_j peuvent ainsi être différentes de est_i et est_j. Néanmoins, l'exécution de cette tâche apériodique ayant lieu avant s_i elle ne peut repousser l'exécution de T_j sans repousser l'exécution de T_i. On a donc la propriété suivante :

$$s_j - s_i \leq est_j - est_i \tag{8.13}$$

Le même raisonnement pouvant être mené pour k tâches apériodiques souples, on obtient donc :

$$(s_j + C_j) - s_i \leq (est_j + C_j) - est_i \leq L_{ij} \tag{8.14}$$

La contrainte de latence est donc dans tous les cas respectée, ce qui prouve le théorème $\boxdot$

Ainsi, si on se restreint au cas où toutes les tâches apériodiques sont souples, il est possible d'attendre en-ligne la date $t = s_i$ pour traduire la contrainte de latence $L(T_i,T_j)$ en une échéance même si au départ, $d_i \neq d_j$. Néanmoins, le résultat suivant montre que cela n'est plus vrai si on considère les tâches apériodiques fermes.

THÉORÈME 8.4
Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ. Si après uniformisation des échéances on a $d_i \neq d_j$, alors ne pas traduire cette contrainte de latence en une échéance avant la date $t = s_i$ peut permettre l'acceptation de tâches apériodiques fermes dont le respect des échéances entraîne le non respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU THÉORÈME 8.4
Puisque l'uniformisation des échéances a préalablement eu lieu, $d_i \neq d_j$ signifie $d_i < d_j$. De plus, $est_i + L_{ij} < d_j$ car le cas échéant cette contrainte de latence aurait été supprimée par les règles de réduction de la section 7.4. Notons I_i et I_j les intervalles d'exécution correspondant à ces échéances. Notons Q_{ij} le nombre maximum de slots pouvant être utilisés à l'exécution de tâches apériodiques entre le début de T_i et la fin de T_j, on peut calculer Q comme suit :

$$Q_{ij} = L_{ij} - \sum_{k=i}^{j} C_k \tag{8.15}$$

Si $sc(I_j) > Q_{ij}$ alors une tâche apériodique ferme $J_a(r_a,C_a,d_a)$ telle que :

$$\begin{cases} r_a < s_i & (A) \\ Q_{ij} < C_a \leq sc(I_j) & (B) \\ d_i < d_a < est_i + L_{ij} & (C) \end{cases} \tag{8.16}$$

est acceptée lorsqu'elle est activée. La quantité de temps disponible dans I_j étant suffisante, il n'est pas indispensable qu'elle débute son exécution avant la fin de la dernière tâche appartenant à I_i que nous appellerons T_k. De plus, de part leurs échéances, T_k ainsi que les tâches la précédant sont plus prioritaires que J_a. Avant que J_a n'ait commencé à s'exécuter, il est donc possible que la tâche T_i s'exécute et définisse une échéance $d_j = s_i + L_{ij}$ pour T_j ainsi que les tâches la précédant. On a donc $d_j > d_a$ ce qui implique que J_a s'exécute avant T_j. On a donc :

$$f_j - s_i \geq \sum_{k=i}^{j} C_k + C_a \tag{8.17}$$

en utilisant l'inéquation (B) on obtient :

$$f_j - s_i > \sum_{k=i}^{j} C_k + Q_{ij} \tag{8.18}$$

en utilisant la définition de Q_{ij}

$$f_j - s_i > L_{ij} \tag{8.19}$$

La contrainte de latence $L(T_i,T_j)$ n'est donc pas respectée, ce qui prouve le théorème □

8.5.1.2 Cas $d_i = d_j$

Nous étudierons d'abord le cas où aucune autre contrainte de latence n'est en inclusion ou chevauche $L(T_i,T_j)$. Ensuite nous traiterons le cas de l'inclusion, puis du chevauchement.

8.5.1.2.1 Cas $\nexists L(T_g,T_h), (i \leq g \leq j) OR (i \leq h \leq j) = TRUE$

On est donc ici dans le cas simple qu'illustre la figure 8.5 page 201. L'hypothèse $d_i = d_j$ et l'uniformisation préalable des échéances implique que $\forall k \quad i < k < j \quad d_k = d_i = d_j$. Soit s_i la date à laquelle sera exécutée pour la première fois la tâche T_i. Sur $]0,s_i]$, trois types de modification de l'ordonnancement hors-ligne peuvent intervenir :

- exécution d'une tâche apériodique souple,
- acceptation d'une tâche apériodique ferme $J_a(r_a,C_a,d_a)$ avec $d_a > d_j$,
- acceptation d'une tâche apériodique ferme $J_a(r_a,C_a,d_a)$ avec $d_a \leq d_j$.

Nous allons à présent prouver le respect de la contrainte de latence pour ces trois types de modification.

LEMME 8.1

Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ telle qu'après uniformisation des échéances on a $d_i \neq d_j$. Si cette contrainte de latence n'est ni en inclusion ni en chevauchement avec une autre, alors l'exécution, selon les règles du slot shifting, de tâches apériodiques souples sur n'importe quel intervalle de temps $]t_1,t_2]$ tel que $0 \leq t_1 < t_2 \leq s_i$ ne peut entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU LEMME 8.1

Preuve en tout point similaire avec celle du théorème 8.3 □

LEMME 8.2
Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ telle qu'après uniformisation des échéances on a $d_i \neq d_j$. Si cette contrainte de latence n'est ni en inclusion ni en chevauchement avec une autre, alors l'acceptation à la date $r_a \leq s_i$, selon les règles du slot shifting, d'une tâche apériodique ferme $J_a(r_a,C_a,d_a)$ avec $d_a > d_j$ ne peut entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU LEMME 8.2
L'acceptation de la tâche apériodique ferme J_a implique qu'elle sera exécutée avant son échéance. Si on se réfère aux règles d'ordonnancement en-ligne du slot shifting énoncées dans 8.2.3.3, sur $]0,s_i]$ cette tâche est moins prioritaire que les tâches ordonnancées hors-ligne et précédant T_j car $d_a > d_j$. Néanmoins elle peut être exécutée pendant un slot à une date t si aucune tâche apériodique souple n'est disponible et si la prochaine tâche périodique T_k à exécuter est telle que $t < est_k$. Dans ce cas seulement elle peut modifier les dates s_i et s_j de la même manière que l'exécution d'une tâche apériodique souple. Le lemme 8.1 permet donc de prouver que J_a ne peut entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$, ce qui prouve le lemme $\boxdot$

LEMME 8.3
Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ telle qu'après uniformisation des échéances on a $d_i \neq d_j$. Si cette contrainte de latence n'est ni en inclusion ni en chevauchement avec une autre, alors l'acceptation à la date $r_a \leq s_i$, selon les règles du slot shifting, d'une tâche apériodique ferme $J_a(r_a,C_a,d_a)$ avec $d_a \leq d_j$ ne peut entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU LEMME 8.3
L'acceptation de la tâche apériodique ferme J_a implique qu'elle sera exécutée avant son échéance. Si on se réfère aux règles d'ordonnancement en-ligne du slot shifting énoncées dans 8.2.3.3, quel que soit l'intervalle temporel considéré sur $]0,s_i]$ cette tâche est plus prioritaire que toutes les tâches périodiques ordonnancées hors-ligne appartenant au même intervalle d'exécution que T_i et T_j. En effet, même en cas d'égalité des échéances, la tâche apériodique est choisie avant toute tâche périodique. Cette tâche sera donc exécutée totalement avant que T_i ne débute son exécution à $t = s_i$. Son exécution avant cette date peut repousser l'exécution de T_i et T_j de la même manière que l'exécution d'une tâche apériodique souple. Or le lemme 8.1 montre que cela ne peut pas entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$, ce qui prouve le lemme $\boxdot$

Les lemmes précédents permettent de montrer que dans le cas présent, il est possible en-ligne de traduire la latence $L(T_i,T_j)$ en une échéance fixe sans nuire au respect de cette contrainte. Ceci est résumé par le théorème suivant :

THÉORÈME 8.5
Soit une contrainte de latence $L(T_i,T_j)$ satisfaite par un ordonnancement hors-ligne σ tel que, après uniformisation des échéances, on a $d_i = d_j$ et que cette contrainte de latence n'est ni en inclusion ni en chevauchement avec une autre. Si cette égalité reste vrai sur un intervalle temporel $]t_1,t_2]$ tel que

$0 \leq t_1 < t_2 \leq s_1$, *alors l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples sur cet intervalle temporel ne peut pas entraîner le non-respect de cette contrainte de latence* $L(T_i,T_j)$.

DÉMONSTRATION DU THÉORÈME 8.5
Le lemme 8.1 prouve que l'exécution des tâches apériodiques souples n'ont pas d'impact sur le respect de la contrainte de latence $L(T_i,T_j)$. *Ensuite, pour chaque tâche apériodique ferme acceptée, on peut appliquer suivant son échéance les résultats du lemme 8.2 ou du lemme 8.3 qui montrent également que cela ne peut entraîner le non-respect de la contrainte de latence* $L(T_i,T_j)$. *Ce qui prouve le théorème* $\boxdot$

Si on applique ce dernier résultat au cas nous intéressant, c'est-à-dire où la contrainte de latence n'est ni en inclusion, ni en chevauchement avec une autre, on obtient le résultat suivant :

THÉORÈME 8.6
Soit une contrainte de latence $L(T_i,T_j)$ *satisfaite par un ordonnancement hors-ligne* σ *tel que, après uniformisation des échéances, on a* $d_i = d_j$. *Si cette contrainte de latence n'est ni en inclusion ni en chevauchement avec une autre, alors l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples avant la date* $t = s_i$ *ne peut pas entraîner le non-respect de cette contrainte de latence* $L(T_i,T_j)$.

DÉMONSTRATION DU THÉORÈME 8.6
Il suffit de prouver que le théorème 8.5 s'applique sur l'intervalle temporel $]0,s_i]$. *Le slot shifting classique ne modifie pas en-ligne les échéances des tâches de l'ordonnancement hors-ligne. La seule possibilité de modification est la traduction en-ligne d'une contrainte de latence* $L(T_g,T_h)$ *en échéance. Une telle modification touche alors toute tâche* T_k *telle que* $g \leq k \leq h$. *Puisque la contrainte de latence* $L(T_i,T_j)$ *n'est en inclusion ou en chevauchement avec aucune autre, on a :*

$$\nexists L(T_g,T_h), (i \leq g \leq j) OR (i \leq h \leq j) = TRUE \tag{8.20}$$

On a donc $d_i = d_j$ *sur l'ensemble de l'intervalle temporel* $]0,s_i]$, *ce qui prouve le théorème* $\boxdot$

8.5.1.2.2 Cas $\exists L(T_g,T_h), i \leq g \leq h \leq j$

On a dans ce cas une contrainte de latence $L(T_g,T_h)$ qui est incluse dans la contrainte de latence $L(T_i,T_j)$. On s'intéresse à la possibilité de ne traduire ces deux contraintes de latence en échéances qu'en-ligne une fois que respectivement T_i et T_g sont exécutées pour la première fois.

THÉORÈME 8.7
Soit deux contraintes de latence $L(T_i,T_j)$ *et* $L(T_g,T_h)$ *satisfaites par un ordonnancement hors-ligne* σ *tel que, après uniformisation des échéances on a* $d_i = d_j = d_g = d_h$. *Si* $L(T_g,T_h)$ *est incluse dans* $L(T_i,T_j)$, *alors l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples avant la date* $t = s_g$ *ne peut pas entraîner le non-respect de ces contraintes de latence.*

DÉMONSTRATION DU THÉORÈME 8.7
Que la contrainte de latence $L(T_g,T_h)$ *soit incluse dans* $L(T_i,T_j)$ *implique que quoi qu'il arrive en-ligne, on aura* $s_i \leq s_g$. *Si* $T_i \neq T_g$, *on a donc deux phases : sur* $]0,s_i]$ *aucune des deux contraintes de*

latence n'est traduite par une échéance fixe, tandis que sur $]s_i,s_g]$ la contrainte de latence $L(T_i,T_j)$ est traduite par une échéance $d_j = s_i + L_{ij}$. Si $T_i = T_g$, les deux contraintes sont traduites par des échéances à $t = s_i = s_g$.

Sur $]0,s_i]$ le théorème 8.5 permet de prouver que l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples sur cet intervalle temporel ne peut pas entraîner le non-respect de ces deux contraintes de latence. Dans le cas $T_i = T_g$, on a $s_i = s_g$ et cela suffit pour prouver le théorème. Dans le cas $T_i \neq T_g$, à la date $t = s_i$, une échéance d_j est créée telle que $d_j = s_i + L_{ij}$, ce qui implique la création d'un nouvel intervalle d'exécution I_j auquel appartiennent toutes les tâches T_k telles que $i \leq k \leq j$, ce qui inclut T_g et T_h. A présent, le respect par la tâche T_j de l'échéance d_j garantit le respect de la contrainte de latence $L(T_i,T_j)$. En ce qui concerne la contrainte de latence $L(T_g,T_h)$, elle n'est plus en inclusion avec aucune autre contrainte de latence et $d_g = d_h$. Il est donc possible d'appliquer le théorème 8.5 pour prouver que sur $]s_i,s_g]$ l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples ne peut pas entraîner le non-respect de la contrainte de latence $L(T_g,T_h)$. Ce qui prouve le théorème □

8.5.1.2.3 Cas $\exists L(T_g,T_h), i < g < j < h$

Enfin, dernier cas, il y a chevauchement de deux contraintes de latence $L(T_i,T_j)$ et $L(T_g,T_h)$ telles que $i < g$. Les deux théorèmes 8.8 et 8.10 montrent que l'impact du chevauchement n'est pas le même pour les deux contraintes de latence.

THÉORÈME 8.8

Soit deux contraintes de latence $L(T_i,T_j)$ et $L(T_g,T_h)$ satisfaites par un ordonnancement hors-ligne σ tel que, après uniformisation des échéances, on a $d_i = d_j = d_g = d_h$. Si $L(T_g,T_h)$ est en chevauchement avec $L(T_i,T_j)$ de manière à ce que $i < g \leq j < h$, alors l'acceptation de tâches apériodiques fermes et l'exécution de tâches apériodiques souples avant la date $t = s_i$ ne peut pas entraîner le non-respect de la contrainte de latence $L(T_i,T_j)$.

DÉMONSTRATION DU THÉORÈME 8.8

Il suffit de prouver que le théorème 8.5 s'applique sur l'intervalle temporel $]0,s_i]$. Le slot shifting classique ne modifie pas en-ligne les échéances des tâches de l'ordonnancement hors-ligne et $i < g$ implique $s_i < s_j$. La contrainte de latence $L(T_g,T_h)$ ne peut donc être traduite en échéance avant la date s_i, ce qui implique que sur l'intervalle temporel $]0,s_i]$ les échéances d_i et d_j restent inchangées. Le théorème 8.5 est donc applicable sur $]0,s_i]$, ce qui prouve le théorème □

Voyons à présent l'impact du chevauchement sur la deuxième contrainte de latence. Celui-ci diffère suivant que l'on se restreint à l'exécution de tâches apériodiques souples ou que l'on inclut l'acceptation de tâches apériodiques fermes.

THÉORÈME 8.9

Soit deux contraintes de latence $L(T_i,T_j)$ et $L(T_g,T_h)$ satisfaites par un ordonnancement hors-ligne σ tel que, après uniformisation des échéances, on a $d_i = d_j = d_g = d_h$. Si $L(T_g,T_h)$ est en chevauchement avec $L(T_i,T_j)$ de manière à ce que $i < g \leq j < h$, alors l'exécution de tâches apériodiques souples avant la date $t = s_g$ ne peut pas entraîner le non-respect de la contrainte de latence $L(T_g,T_h)$.

DÉMONSTRATION DU THÉORÈME 8.9
Avant l'exécution, on a $d_g = d_h$, ce qui reste vrai tant que la contrainte $L(T_i,T_j)$ n'est pas traduite en échéance à la date $t = s_i$. Sur l'intervalle temporel $]0,s_i]$ on peut donc appliquer le théorème 8.5 à la contrainte de latence $L(T_g,T_h)$. A la date $t = s_i$, l'échéance correspondant à la contrainte de latence $L(T_i,T_j)$ est créée. Comme $i < g \leq j < h$ l'échéance d_g est modifiée alors que d_h ne l'est pas. On a donc $d_g \neq d_h$ sur l'intervalle temporel $]s_i,s_g]$ et on peut appliquer le théorème 8.3 à la contrainte de latence $L(T_g,T_h)$ sur cet intervalle temporel. L'exécution de tâches apériodiques souples sur les intervalles temporels $]0,s_i]$ et $]s_i,s_g]$ ne peut donc pas pas entraîner le non-respect de la contrainte de latence $L(T_g,T_h)$, ce qui prouve le théorème.

Il apparaît donc que si on se restreint à l'exécution de tâches apériodiques souples, la traduction des deux contraintes de latence se chevauchant est possible en-ligne sans que cela nuise au respect de ces échéances. Comme le montre le résultat suivant, ce n'est plus le cas si on inclut l'acceptation de tâches apériodiques fermes.

THÉORÈME 8.10
Soit deux contraintes de latence $L(T_i,T_j)$ et $L(T_g,T_h)$ satisfaites par un ordonnancement hors-ligne σ tel que, après uniformisation des échéances, on a $d_i = d_j = d_g = d_h$. Si $L(T_g,T_h)$ est en chevauchement avec $L(T_i,T_j)$ de manière à ce que $i < g \leq j < h$, alors ne pas traduire la contrainte de latence $L(T_g,T_h)$ en une échéance avant la date $t = s_g$ peut permettre l'acceptation de tâches apériodiques fermes dont le respect des échéances entraîne le non respect de la contrainte de latence $L(T_g,T_h)$.

DÉMONSTRATION DU THÉORÈME 8.10
A la date $t = s_i$, l'échéance correspondant à la contrainte de latence $L(T_i,T_j)$ est créée et comme $i < g \leq j < h$ l'échéance d_g est modifiée alors que d_h ne l'est pas. On a donc $d_g \neq d_h$ sur l'intervalle temporel $]s_i,s_g]$ et on peut appliquer le théorème 8.4 à la contrainte de latence $L(T_g,T_h)$ sur cet intervalle temporel, ce qui prouve le théorème. $\boxdot$

8.5.1.3 Construction de $\mathcal{L}_{hl}$

Le tableau 8.2 résume les résultats précédents.

tâches apériodiques	hors-ligne $d_i \neq d_j$	hors-ligne $d_i = d_j$				
		sans chevauchement sans inclusion	inclusion $i \leq g \leq h \leq j$		chevauchement $i < g \leq j < h$	
			$L(T_i,T_j)$	$L(T_g,T_h)$	$L(T_i,T_j)$	$L(T_g,T_h)$
souples	oui	oui	oui	oui	oui	oui
fermes	non	oui	oui	oui	oui	non

TAB. 8.2 – *Tableau récapitulant s'il est possible de traduire une contrainte de latence en une échéance en-ligne sans nuire au respect de cette contrainte*

Il est donc possible de discerner les contraintes de latence pouvant être traduites en échéances en-ligne et celles qui ne le peuvent pas, constituant ainsi $\mathcal{L}_{hl}$. Notons néanmoins que si on se restreint au cas des tâches apériodiques souples seulement (pas d'acceptation de tâches apériodiques

fermes), toutes les les contraintes de latence peuvent être traduites en échéances en-ligne et $\mathcal{L}_{hl} = \emptyset$. Le cas échéant, on construit $\mathcal{L}_{hl}$ en utilisant l'algorithme 7. Celui-ci nécessite que l'ordonnancement σ soit donné sous forme d'un tableau dans lequel l'indice des lignes corresponde à l'indice des tâches (et donc à leur ordre d'exécution). Chaque ligne donne pour chaque tâche sa durée et sa date de début (inutilisée ici), son échéance, la liste $\mathcal{L}_s$ des contraintes de latence dont elle est la première tâche (T_i d'une contrainte de latence $L(T_i,T_j)$) et la liste $\mathcal{L}_f$ des contraintes de latence dont elle est la deuxième tâche (T_j d'une contrainte de latence $L(T_i,T_j)$). Les listes $\mathcal{L}_s$ sont ordonnées par indices j décroissants des contraintes $L(T_i,T_j)$ tandis que les listes $\mathcal{L}_f$ sont ordonnées par indices i décroissants des contraintes $L(T_i,T_j)$.

Algorithme 7 de construction de $\mathcal{L}_{hl}$

1: $\mathcal{L}_{ouvertes} = \emptyset$ est une suite ordonnée
2: $\mathcal{L}_{hl} = \emptyset$ est l'ensemble solution retourné par l'algorithme
3: **pour** chaque tâche T_k du tableau, k allant de 1 à N **faire**
4: **pour** chaque contrainte de latence $L(T_i,T_j)$ appartenant à $\mathcal{L}_s$ **faire**
5: $L(T_i,T_j)$ est ajouté à $\mathcal{L}_{ouvertes}$
6: **fin pour**
7: **pour** chaque contrainte de latence $L(T_i,T_j)$ appartenant à $\mathcal{L}_f$ **faire**
8: **si** $d_i \neq d_j$ **alors**
9: $L(T_i,T_j)$ est ajouté à $\mathcal{L}_{hl}$
10: **fin si**
11: Toutes les contraintes de latences placées après $L(T_i,T_j)$ dans $\mathcal{L}_{ouvertes}$ sont ajoutées à $\mathcal{L}_{hl}$
12: $L(T_i,T_j)$ est supprimé de à $\mathcal{L}_{ouvertes}$
13: **fin pour**
14: **fin pour**

En fonction des contraintes de latence composant l'ensemble $\mathcal{L}_{hl}$ construit par l'algorithme 7, on modifie ensuite les échéances de certaines tâches comme suit :

$$\forall L(T_i,T_j) \in \mathcal{L}_{hl} \quad d_j = est_i + L_{ij} \tag{8.21}$$

Enfin, on applique à nouveau l'uniformisation des échéances.

8.5.2 Calculs en-ligne pour l'acceptation de tâches apériodiques

Traduire en-ligne une contrainte de latence $L(T_i,T_j)$ en échéance ne peut se faire uniquement une fois que l'ordonnancement en-ligne a choisi d'exécuter, pour la première fois, la tâche T_i dans le prochain slot. C'est donc lors de la mise à jour des spare capacities qui suit cette décision qu'il faut intervenir. Le test d'acceptation des tâches apériodiques fermes, les choix effectués lors de l'ordonnancement en-ligne restent ceux décrits dans 8.2.3.2 et 8.2.3.3.

Le problème que nous nous sommes posé est celui de l'exécution en-ligne de tâches apériodiques souples et fermes. Nous détaillerons donc tout d'abord les calculs en-ligne nécessaires dans ce cas. Néanmoins la sous-section précédente ayant fait apparaître des résultats intéressants dans le cas où on se restreint à des tâches apériodiques souples, on donne ensuite une variante de ces calculs en-ligne pour ce cas plus restrictif.

8.5.2.1 Acceptation de tâches apériodiques fermes et exécution de tâches apériodiques souples

Nous avons vu que si on considère une contrainte de latence $L(T_i,T_j)$ qui n'a pas été traduite hors-ligne par une échéance, on a toujours $d_i = d_j$ à la date s_i.

Il se peut tout d'abord qu'à $t = s_i$ la création d'une échéance pour traduire la contrainte de latence $L(T_i,T_j)$ soit devenue inutile. C'est le cas si :

$$d_j \leq s_i + L_{ij} \tag{8.22}$$

Bien que les règles de réduction présentées à la section 7.4 page 182 suppriment les contrainte de latence $L(T_p,T_q)$ telles que $d_q \leq est_p + L_{pq}$, il se peut que la date de début d'exécution de la tâche T_i ait été repoussée d'une durée suffisante pour que la contrainte de latence $L(t_i,T_j)$ soit devenue moins contraignante que l'échéance initiale.

Le cas échéant, si $d_j > s_i + L_{ij}$ il faut modifier d_j comme suit :

$$d_j = s_i + L_{ij} \tag{8.23}$$

ce qui définit un nouvel intervalle d'exécution sauf s'il existe une tâche apériodique ferme préalablement acceptée et possédant la même échéance.

Considérons d'abord le cas où un nouvel intervalle d'exécution est créé. Soit I_p l'intervalle d'exécution auquel appartenaient les tâches T_k telles que $i \leq k \leq j$ avant traduction de la contrainte de latence $L(t_i,T_j)$ et soit I_e l'intervalle d'exécution tel que $d_j \in I_d$. On a forcément $e \leq p$. Les tâches T_k telles que $i \leq k \leq j$ appartiennent au nouvel intervalle d'exécution créé I_j correspondant à d_j. On incrémente donc la spare capacity de I_p de la somme des durées d'exécution de ces tâches :

$$sc(I_p) = \begin{cases} sc_{old}(I_p) + \sum_{k=i}^{j} C_k & \text{si } I_p \neq I_e \\ sc_{old}(I_p) + \sum_{k=i}^{j} C_k - |I_d| & \text{si} I_p = I_e \end{cases} \tag{8.24}$$

Si $I_p \neq I_e$, les spare capacities des intervalles d'exécution I_k tels que $e < k < p$ sont ensuite mises à jour en utilisant la formule classique du slot shifting [9] (8.3 page 192). Puis, la spare capacity de l'intervalle d'exécution I_e est calculée en soustrayant à l'ancienne valeur la taille du nouvel intervalle d'exécution créé $|I_d| = d_j - start(I_e)$.

L'intervalle d'exécution est ensuite créé, les tâches T_k telles que $i \leq k \leq j$ lui appartiennent et on utilise la formule 8.3 page 192 pour calculer sa spare capacity, puis mettre à jour celles des intervalles d'exécution le précédant jusqu'à atteindre l'intervalle d'exécution courant, ou vérifier la condition d'arrêt 8.5 page 195.

9. il est possible d'utiliser la condition d'arrêt 8.5 page 195.

S'il existe une tâche apériodique ferme préalablement acceptée et ayant pour échéance d_j, l'intervalle d'exécution existe déjà et il n'est pas nécessaire de retrancher $|I_d|$ à la spare capacity de l'intervalle d'exécution I_e.

Il se peut qu'au début de l'exécution de la tâche T_i il y ait plusieurs échéances à traduire, par exemple s'il existe deux contraintes de latence $L(T_i,T_j)$ et $L(T_i,T_k)$. On traduira alors les contraintes par ordre décroissant d'indice de leur deuxième tâche (j pour une contrainte de latence $L(T_i,T_j)$). Cela assure [10] que pour chaque contrainte de latence $L(T_i,T_j)$, au moment où elle est traduite, on a encore $d_i = d_j$.

8.5.2.2 Exécution de tâches apériodiques souples seulement

Si on se restreint aux tâches apériodiques souples, aucune contrainte de latence n'a été traduite en échéance fixe hors-ligne, mais l'hypothèse que pour chaque contrainte de latence $L(T_i,T_j)$ traduite en-ligne on a $d_i = d_j$ jusqu'à la date $t = s_i$ ne tient plus. Cela complique quelque peu la création du nouvel intervalle d'exécution et la mise à jour des spare capacities.

En effet, toute les tâches T_k telles que $i \leq k \leq j$ n'appartiennent pas, d'une part, forcément au même intervalle d'exécution, et d'autre part certaines d'entre elles possèdent peut-être déjà une échéance plus contraignante que celle nouvellement créée.

Néanmoins, par construction on a forcément [11] :

$$\forall a,b \quad i \leq a < b \leq j \Rightarrow d_a \leq d_b \tag{8.25}$$

On a donc pour l'ensemble des tâches T_k telles que $i \leq k \leq j$ différents cas de figure :

- $d_j \leq s_i + L_{ij}$: traduire la contrainte de latence $L(T_i,T_j)$ est inutile car l'échéance équivalente serait moins contraignante que celles déjà assignées au tâches,
- $d_i = d_j$ et $d_j > s_i + L_{ij}$: on peut appliquer la même mise à jour que pour le cas d'exécution de tâches apériodiques souples et fermes (voir 8.5.2.1),
- $d_i \leq s_i + L_{ij}$ et $d_i \neq d_j$: toutes les tâches T_k telles que $i \leq k \leq j$ vont changer d'échéance mais appartiennent pour l'instant à des intervalles d'exécution différents,
- $d_i \leq s_i + L_{ij} < d_j$: parmi les tâches T_k telles que $i \leq k \leq j$ certaines vont garder leur échéance et d'autres vont changer et peuvent appartenir pour l'instant à des intervalles d'exécution différents.

La mise à jour n'est donc pas si différente du cas d'exécution de tâches apériodiques souples et fermes, si ce n'est qu'il n'existe pas forcément un seul intervalle d'exécution I_p auquel appartiennent toutes les tâches T_k telles que $i \leq k \leq j$. Si on note I_q l'intervalle d'exécution auquel appartient T_j, et I_e l'intervalle d'exécution contenant la date $t = s_i + L_{ij}$, il est nécessaire de mettre à jour les spare capacities des intervalles d'exécution des intervalles I_q à I_e de droite à gauche avec

10. Cela est dû à une des règles de réduction présentée à la section 7.4 concernant les cas d'inclusion de contraintes de latence.

11. Preuve évidente. Cela est vrai dans l'ordonnancement hors-ligne à cause de l'uniformisation des échéances. En-ligne, ni T_a ni T_b n'ont été exécutées puisqu'on se place à la date $t = s_i$ et que $i \leq a < b$. Or la traduction de contrainte de latence en-ligne ne peut avoir modifié l'échéance de T_b sans que celle de T_a soit modifiée de la même manière ou qu'elle soit déjà inférieure.

la formule 8.3 page 192, les tâches T_k telles que $i \leq k \leq j$ ne leur appartenant plus. On retranche à la spare capacity de I_e la taille $|I_j|$ de l'intervalle d'exécution créé par la traduction de la contrainte de latence $L(T_i,T_j)$. Changent d'échéance et appartiennent donc à cet intervalle d'exécution toutes les tâches T_k telles que $i \leq k \leq j$ et $d_k > s_i + L_{ij}$. On utilise alors la formule 8.3 page 192 pour calculer la spare capacity de ce nouvel intervalle d'exécution puis mettre à jour celles des intervalles d'exécution le précédant jusqu'à atteindre l'intervalle d'exécution courant, ou vérifier la condition d'arrêt 8.5 page 195.

8.5.3 Complexité

On différencie le cas où l'on considère conjointement tâches apériodiques souples et fermes, du cas où l'on se restreint à des tâches apériodiques souples uniquement.

8.5.3.1 Acceptation de tâches apériodiques fermes et exécution de tâches apériodiques souples

Hors-ligne, la différence est la construction de $\mathcal{L}_{hl}$ qui nécessite un parcours de l'ordonnancement de complexité $O(N)$. Durant ce parcours, chaque contrainte de latence $L(T_i,T_j)$ entraîne un traitement une fois ce parcours arrivé à la tâche T_i et à la tâche T_j. Cela donne une complexité théorique en $O(N+2L)$. D'autre part la modification des échéances a lieu deux fois, ce qui rajoute un parcours en $O(N)$. La complexité ($O(N+2L)$) des calculs hors-ligne qu'implique la prise en compte des contraintes de latence dans le slot shifting est donc très raisonnable.

En-ligne, une nouvelle mise à jour a lieu lorsqu'une contrainte de latence $L(T_i,T_j)$ est traduite en échéance, c'est-à-dire quand T_i est choisie la première fois pour être exécutée dans le prochain slot. Cette mise à jour implique la création d'un nouvel intervalle d'exécution et le recalcul des spare capacities d'un ensemble d'intervalles d'exécution. Cette mise à jour est en $O(I)$ où I est le nombre d'intervalles d'exécution. Elle est en tout point équivalente à la mise à jour effectuée après acceptation d'un tâche apériodique ferme (voir 8.2.3.2 page 194). Nous avons en outre présenté précédemment une condition d'arrêt (8.5 page 195) qui permet de réduire le nombre d'intervalles d'exécution à mettre à jour. Il peut néanmoins y avoir à une date t traduction de k contraintes de latence et donc complexité en $O(kI)$. Cela est équivalent dans le slot shifting classique à la complexité des tests d'acceptation de k tâches apériodiques fermes à une même date. Or s'il est possible pour un ordonnancement σ de borner k comme étant le nombre maximum de contraintes de latence ayant la même première tâche (tâche T_i d'une contrainte de latence $L(T_i,T_j)$), le nombre de tâches apériodiques pouvant être acceptées à une date t n'est pas bornée. La complexité des calculs en-ligne n'est donc pas augmentée.

8.5.3.2 Exécution de tâches apériodiques souples seulement

Il n'y a dans ce cas aucun traitement supplémentaire à effectuer hors-ligne.

En-ligne, il faut comparer cette complexité à celle du slot shifting réduit à l'exécution de tâches apériodiques souples seulement. Les calculs en-ligne consistent alors à l'ordonnancement en-ligne (exécution d'une tâche périodique ou apériodique souple) puis à la mise à jour des spare capacities qui s'en suit.

L'ordonnancement en-ligne nécessite le test de la spare capacity de l'intervalle d'exécution courant : si elle est supérieure à zéro et qu'une tâche apériodique souple est disponible on l'exécute, sinon on exécute un tâche périodique, si disponible. L'ordonnancement en-ligne peut donc être considéré comme faisable en temps constant puisque ne variant pas en fonction des grandeurs du problèmes (nombre de tâches périodiques et apériodiques). Cette complexité ne varie pas lorsqu'on introduit les contraintes de latence dans le slot shifting.

La mise à jour des spare capacities qui suit l'ordonnancement est en $O(I)$. Comme nous l'avons vu dans 8.5.3.1, la traduction de contraintes de latence qui précède cette mise à jour est elle en $O(kI)$ où k est le nombre de contraintes de latence traduites à une même date. Il y a donc augmentation de la complexité théorique qui égale celle du slot shifting classique avec acceptation de tâches apériodiques fermes. Néanmoins, pour chaque ordonnancement σ, k est borné alors que le nombre de tâches apériodiques fermes pouvant être acceptées ne l'est pas. La complexité de la prise en compte des contraintes de latence dans le slot shifting limité à l'exécution de tâches apériodiques souples est donc inférieure à celle du slot shifting classique avec acceptation de tâches apériodiques fermes. La complexité est donc en $O(kI)$ avec k borné pour un ordonnancement hors-ligne donné.

8.5.4 Optimalité

Nous avons vu que dans le cas d'acceptation des tâches apériodiques fermes, on était obligé de distinguer deux catégories de contraintes de latence. Les premières sont traduites en-ligne sans sur-contraindre le système puisqu'on attend qu'elles correspondent à des échéances fixes. Les secondes qui appartiennent à $\mathcal{L}_{hl}$ sont traduites hors-ligne en échéances fixes qui sur-contraignent l'ordonnancement.

Il n'est donc pas possible si $\mathcal{L}_{hl} \neq \emptyset$ d'atteindre l'optimalité dans l'ordonnancement mixte de tâches périodiques avec contraintes d'échéance et de latence et de tâches apériodiques fermes. Par contre celle-ci est atteinte si $\mathcal{L}_{hl} = \emptyset$ car la traduction des contraintes de latence en-ligne ne sur-contraint pas l'ordonnancement et l'optimalité du slot shifting classique (voir 8.2.4 page 200) est alors conservée. Nous rappelons que $\mathcal{L}_{hl} = \emptyset$ si, dans l'ordonnancement hors-ligne après uniformisation et réduction, il n'y a aucune contrainte de latence qui en chevauche une autre et que pour toute contrainte de latence $L(T_i,T_j)$, on a $d_i = d_j$.

Lorsqu'on se restreint à l'exécution de tâches apériodiques souples, nous avons montré que toutes les contraintes de latence peuvent être traduites en-ligne sans sur-contraindre l'ordonnancement. Traduire toutes les contraintes de latence en échéances fixes en-ligne permet donc d'élargir l'optimalité du slot shifting pour l'ordonnancement mixte de tâches périodiques avec échéances et de tâches apériodiques souples au cas où les tâches périodiques possèdent également des contraintes de latence.

Conclusion de la partie II

L'état de l'art a montré qu'il existait de nombreux travaux sur l'ordonnancement mixte de tâches périodiques et apériodiques. Néanmoins, ils s'appliquent tous au modèle temps réel classique restreint aux contraintes d'échéance et font souvent des hypothèses simplificatrices (notamment que l'échéance est égale à la période). Les méthodes de type serveur perdent par exemple tout intérêt dès que l'hypothèse de la période égale à l'échéance est levée puisqu'il devient alors hasardeux de dimensionner le serveur.

De plus, si la complexité du problème d'implantation temps réel distribuée sur architecture hétérogène implique une approche hors-ligne, il est possible dans les heuristiques employées hors-ligne de prendre en compte davantage de contraintes temps réel. En effet, le succès du modèle temps réel classique s'explique par sa simplicité et, de ce fait, par l'existence d'algorithmes simples d'ordonnancement pouvant être implémentés sous forme d'ordonnanceur et exécutés en-ligne. La contrepartie de cette simplicité est l'impossibilité de définir des relations temporelles entre plusieurs tâches. La contrainte de latence permet de spécifier une durée maximale pouvant séparer le début de l'exécution d'une tâche de la fin d'exécution d'une autre. Parce qu'elle est relative à une date qui n'est fixée qu'en ligne (date de début d'exécution d'une tâche), la contrainte temporelle qu'elle définit ne correspond pas à une date fixe. Il était donc intéressant d'étudier comment cette liberté peut être exploitée pour exécuter des tâches apériodiques en plus des tâches de l'ordonnancement hors-ligne possédant des contraintes de latence. Pour cela, nous sommes partis de l'hypothèse qu'un ordonnancement hors-ligne, respectant des contraintes d'échéance et de latence, avait auparavant été calculé.

Les calculs en-ligne nécessaires à l'acceptation de tâches apériodiques fermes et à l'exécution de tâches apériodiques fermes et souples doivent être de faible complexité pour, à défaut d'en négliger le coût, être bornés et pris en compte dans les durées d'exécution. Or la complexité de ces calculs est directement liée au nombres de contraintes à satisfaire, qu'il s'agisse d'échéances ou de contraintes de latence. C'est pourquoi nous avons d'abord proposé des règles de réduction permettant de garantir toutes les contraintes de l'ordonnancement hors-ligne en n'en considérant en-ligne qu'un sous-ensemble.

Ensuite nous avons choisi d'étendre la méthode de slot shifting aux contraintes de latence. Notre choix a été motivé par le fait que le slot shifting permet de prendre en compte des tâches apériodiques souples et fermes. Son test d'acceptation est en outre de complexité plus réduite que celle des autres méthodes. Enfin, il est optimal aussi bien pour l'acceptation de tâches apériodiques fermes que pour les temps de réponse des tâches apériodiques souples. Le slot shifting reposant sur la notion d'intervalles d'exécution dont les bornes sont les échéances, nous avons proposé deux

approches pour traduire les contraintes de latence en échéances.

La première consiste à traduire hors-ligne les contraintes de latence en échéances. En-ligne des calculs supplémentaires sont nécessaires pour gérer la mobilité des échéances résultant des contraintes de latence. Cette approche permet d'étendre l'optimalité du slot shifting pour l'ordonnancement mixte de tâches périodiques avec contraintes d'échéance et de tâches apériodiques fermes et souples au cas où les tâches périodiques comportent également des contraintes de latence. Cette approche n'est néanmoins pas utilisable en pratique car la complexité des calculs en-ligne est trop fortement augmentée passant de $O(I)$ à $O(N^2)$ où I est le nombre d'intervalles d'exécution et N le nombre de tâches de l'ordonnancement hors-ligne ($I \leq N$).

La seconde consiste à traduire les contraintes de latence en-ligne lorsqu'il est possible de les remplacer par des échéances fixes, ce qui permet de s'affranchir des calculs nécessaires à la mobilité. Si on se restreint à l'ordonnancement mixte de tâches périodiques avec contraintes d'échéance et de latence et de tâches apériodiques souples, alors toutes les contraintes de latence peuvent être traduites en-ligne. L'optimalité en termes de temps de réponse des tâches apériodiques souples est dans ce cas atteinte. La complexité augmente par rapport au slot shifting classique mais reste acceptable, passant de $O(I)$ à $O(kI)$ où k est le nombre maximum de contraintes de latence à traduire en-ligne à une même date. Si par contre on inclut l'acceptation de tâches apériodiques fermes, nous avons montré que la traduction en-ligne d'une contrainte de latence n'est possible que dans certains cas de figure dépendant à la fois des échéances des tâches et des relations entre contraintes de latence. Certaines contraintes ne pouvant être traduites en-ligne, elles sont traduites hors-ligne en échéance fixes qui sur-contraignent l'ordonnancement. Si de telles relations entre les contraintes de latence existent il n'est alors pas possible d'atteindre l'optimalité. Le cas échéant l'optimalité du slot shifting classique est étendue au cas où les tâches périodiques possèdent également des contraintes d'échéances. L'intérêt de cette approche est qu'elle n'augmente pas la complexité du slot shifting classique, ce qui la rend utilisable en pratique à défaut d'être optimale dans tous les cas.

En utilisant cette deuxième approche, la flexibilité des contraintes de latence peut ainsi être exploitée en-ligne avec un surcoût faible. Cela permet d'obtenir des temps de réponse plus courts et un plus grand taux d'acceptation des tâches apériodiques que dans le cas pour un ordonnancement hors-ligne ne comportant des échéances à la place des contraintes de latence.

Conclusion générale et perspectives

Cette thèse s'inscrit dans le domaine de l'implantation distribuée des systèmes temps réel. Nos objectifs concernent deux étapes de l'implantation distribuée des systèmes temps réel. La première étape correspond aux méthodes nécessaires pour obtenir une implantation distribuée de programmes conditionnés. La deuxième étape consiste, quand à elle, à exécuter conjointement les fonctionnalités prévues hors-ligne avec d'autres fonctionnalités imprévues, tout en respectant les contraintes temps réel. Les apports relatifs à ces deux problématiques devaient ainsi permettre de faciliter l'implantation des systèmes distribués temps réel.

Nous avons illustré nos apports par la figure 8.9 page 226. Celle-ci est à comparer avec la figure 1 page 19 qui en introduction présentait les objectifs de la thèse. En ce qui concerne l'implantation distribuée de programmes conditionnés (partie encadrée en pointillé à droite de la figure), nous avons proposé un modèle flot de données conditionné. Il permet, par sa mise à plat, de modifier automatiquement les programmes de manière à distribuer le conditionnement qu'ils contiennent. Comme notre modèle, bien qu'il facilite l'implantation distribuée, n'est pas le mieux adapté à la description du conditionnement et à la simulation, nous avons proposé deux traductions respectivement de langage FSM (souvent utilisé pour le conditionnement) et de langage schéma-bloc (utilisé pour la simulation) en langage SynDEx qui utilise notre modèle. Une fois le programme traduit, on peut utiliser les heuristiques du logiciel SynDEx, logiciel de CAO niveau système, pour l'implanter sur une architecture distribuée.

Ce modèle flot de données conditionné est donc à présent utilisé par le logiciel SynDEx. De plus, une chaîne de développement complète allant de la simulation avec Scicos à la génération de code multi-processeur avec SynDEx existe grâce à la traduction Scicos/SynDEx. De même la traduction de SyncCharts en SynDEx a été validée par une implémentation des principes de base. Enfin, des exemples de systèmes industriels ont par ailleurs été développés à l'aide de ces traductions et de SynDEx, validant ainsi l'approche proposée.

En ce qui concerne l'ordonnancement mixte hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques (partie inférieure encadrée en pointillés sur la figure), notre apport concerne deux points. Une fois qu'un ordonnancement hors-ligne respectant des contraintes d'échéance et de latence a été calculé, nous avons d'abord présenté des règles de réduction du nombre de contraintes. Ces règles permettent de garantir toutes les contraintes de l'ordonnancement hors-ligne en n'en considérant en-ligne qu'un sous-ensemble. Ensuite nous avons étendu la méthode du slot shifting classique au cas où les tâches périodiques possèdent des contraintes de latence. Nous avons proposé deux versions, l'une optimale mais possédant une complexité la rendant non utilisable en pratique, et une autre version ne modifiant pas

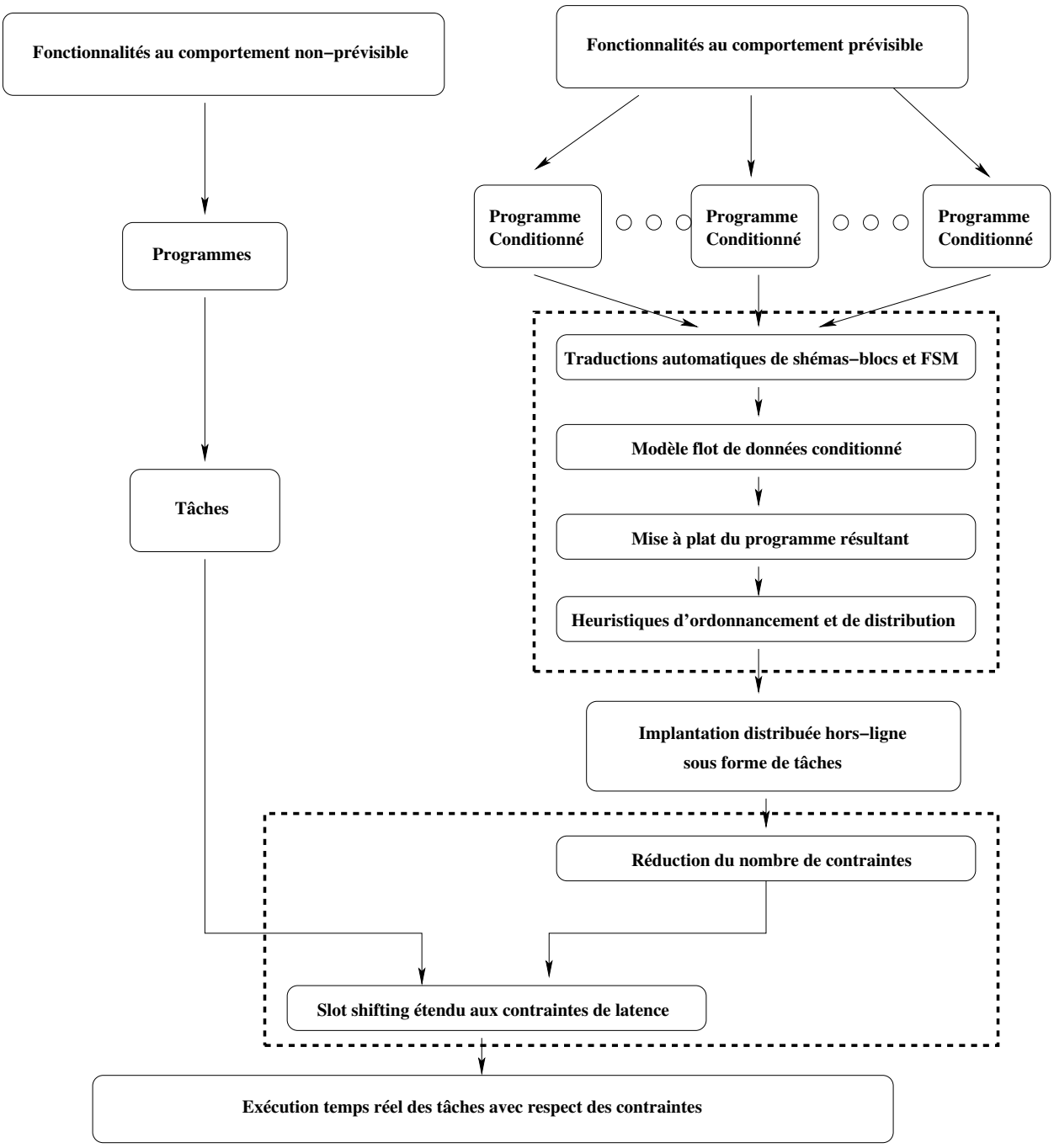

FIG. 8.9 – *Apports de la thèse relativement à l'implantation distribuée des systèmes temps réel*

la complexité mais qui n'est optimale que sous certaines conditions.

Dans les travaux futurs, il reste à évaluer quantitativement les résultats obtenus par cette deuxième version, par rapport au slot shifting classique. En soumettant dans les deux cas un même ordonnancement hors-ligne au même flux de tâches apériodiques, il faudra comparer les résultats obtenus en termes d'acceptation de tâches apériodiques et de temps de réponse.

En outre, des travaux sont en cours pour adapter les heuristiques d'ordonnancement et de distribution de SynDEx aux contraintes de latence. On pourra alors utiliser notre version étendue du slot shifting pour, en plus d'exécuter l'ordonnancement produit hors-ligne par le logiciel, accepter et exécuter des tâches apériodiques. Les calculs hors-ligne et en-ligne nécessaires pourront, par ailleurs, être inclus dans les codes exécutables générés automatiquement par SynDEx.

Par ailleurs, dans les résultats énoncés dans la seconde partie de la thèse, l'ordonnancement hors-ligne est supposé sans alternatives, une version pessimiste de l'ordonnancement comportant des alternatives étant utilisée. Il sera intéressant d'étudier les résultats que l'on peut obtenir en ordonnancement mixte de tâches périodiques et apériodiques lorsque l'on considère les différentes alternatives des ordonnancements hors-ligne pour l'acceptation de tâches apériodiques. Il faudra certainement faire un compromis entre optimalité et complexité acceptable en-ligne.

Enfin, nous avons ici associé périodique à temps réel strict et apériodique à temps réel souple. L'état de l'art a montré que cette représentation était répandue. Néanmoins, la réalité est moins tranchée, des contraintes temps réel strict pouvant être, moyennant quelques hypothèses, associées à des tâches apériodiques. Le déclenchement d'un signal de sécurité devant entraîner l'arrêt d'une partie du système est un exemple d'une telle fonctionnalité. Losqu'une telle tâche apériodique demande son exécution, il sera intéressant de montrer comment l'accepter sans réserve et de chercher à minimiser l'impact de son exécution sur le reste de l'ordonnancement.

Bibliographie

[1] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[2] G. Fohler. *Flexibility in Statically Scheduled Hard Real-Time Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1994.

[3] H. Chetto and M. Chetto. An optimal algorithm for guaranteeing sporadic tasks in hard real-time systems. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, 1990.

[4] X. Liu, J. Liu, J. Eker, and E. A. Lee. Heterogeneous modeling and design of control systems. In *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley-IEEE Press, Tariq Samad and Gary Balas edition, April 2003.

[5] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publichers, 1997.

[6] A.M. Turing. On computable numbers, with an application to the entscheindungs problem. In *Proc. London Math. Soc.*, 1936.

[7] R. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7):424–436, 1979.

[8] J. von Neumann. First draft of a report on the EDVAC. Contract no. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia, june 1945. Reprinted (in part) in Randell, Brian. 1982. Origins of Digital Computers: Selected Papers, Springer-Verlag, Berlin Heidelberg, pp. 383-392.

[9] R. Floyd. The paradigms of programming. *Comm. ACM*, 22(8):455–460, August 1979.

[10] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Publishers, Dordrecht Boston, 1993.

[11] P. Le Guernic, I. Gautier, M. Leborgne, and C. Lemaire. Programming real-time applications with Signal. In *IEEE, Proceedings*, volume 79, pages 1321–1336, September 1991.

[12] F. Boussinot and R. De Simone. The Esterel language. In *Proceedings of the IEEE*, pages 1293–1304, September 1991.

[13] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages*, pages 178–188, Munich, Germany, January 1987.

[14] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow language Lustre. In *SIGSOFT '91: Proceedings of the conference on Software for citical systems*, pages 112–119, 1991.

[15] A. Bouali. XEVE, an Esterel verification environment. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504. Springer-Verlag, 1998.

[16] M. Le Borgne, H. Marchand, E. Rutten, and M. Samaan. Formal verification of Signal programs: Application to a power transformer station controller. In *AMAST '96: Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, pages 271–285. Springer-Verlag, 1996.

[17] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guerniç, and R. De Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE, Special issue on modeling and design of embedded systems*, volume 91, pages 64–83, 2003.

[18] A.L. Ambler, M.M. Burnett, and B.A. Zimmerman. Operational versus definitional: a perspective on programming paradigms. *Computer*, 25(9):28–43, 1992.

[19] D. McCracken. *A guide to Fortran programming*. Wiley, 1961.

[20] A. Goldberg and D. Robson. *Smaltalk-80: the langage and its implementation*. Addison-Wesley, Reading, Mass., 1983.

[21] J. Backus. Can programming be liberated from the Von Neumann style ? A functionnal style and its algebra of programms. *Commun. ACM*, 21(8):613–641, August 1978.

[22] P. Hudak, S. Peyton Jones, and Wadler P. Report on the programming language Haskell, a nonstrict purely functional language version 1.2. *ACM SIGPlan Notices*, 27(5), may 1992.

[23] P. Weis and X. Leroy. *Le langage Caml*. Dunod, 1999.

[24] G.H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[25] E. Moore, editor. *Sequential machines*, chapter Selected papers. Addison-Wesley Publishing Co., Redwood City, CA, 1964.

[26] A. Girault, B. Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design*, 18(6):742–760, June 1999. Research report UCB/ERL M97/57.

[27] R. David and H. Alla. *Petri Nets and Grafcet: tools for modeling of discrete event systems*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[28] R. David. Grafcet: a powerful tool for the specification of logic controllers. *IEEE Trans. Contr. Syst. Tech.*, 3:253–268, 1995.

[29] D. Harel. Statecharts: a visual formalism for complex systems. In *Science of Computer Programming*, volume 8, pages 231–274, 1987.

[30] G. Berry. *The Foundations of Esterel*. MIT Press, 1998.

[31] F. Maraninchi and Y. Rémond. Argos: an automaton-based synchronous language. *Computer Languages*, 27(1-3):61–92, 2001.

[32] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96 IEE-SMC*, Lille, France, July 1996.

[33] R.M. Karp and R.E. Miller. Properties of a model parallel computation: determinacy, terminations, queueing. *SIAM J. Appl. Math.*, 14:1390–1411, November 1966.

[34] J.B. Dennis. First version of a dataflow procedure language. In *Lecture Notes in Computer Sci.*, volume 19, pages 362–376. Springer-Verlag, 1974.

[35] Y. Sorel. Massively parallel systems with real time constraints, the algorithm architecture adequation methodology. In *Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, May 1994.

[36] R. Nikoukhah and S. Steer. Scicos – A dynamic system builder and simulator. In *Proceedings of the 1996 IEEE International Symposium on Computer-Aided Control System Design*, September 1996.

[37] J.T. Buck and E.A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Proc. in the IEEE ICASSP*, Minneapolis, 1993.

[38] W.B. Ackerman and J.B. Dennis. Val–a value-oriented algorithmic language preliminary reference manual. Technical Report 218, Laboratory for Computer Science, MIT, Cambridge, MA, June 1979.

[39] Arvind and R.E. Thomas. I-structures: an efficient data structure for functional languages. Technical report mit/lcs/tm-178, Laboratory for Computer Science, MIT, Cambridge, MA, April 1981.

[40] Arvind and D.E. Culler. Dataflow architecture. *Annual Review in Computer Science*, 1:225–253, 1986.

[41] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. In *IEEE, Proceedings*, volume 79, pages 1305–1320, September 1991.

[42] E. Rutten and P. Le Guernic. Sequencing data flow tasks in Signal. Technical Report Research Report No. 2210, INRIA, November 1993.

[43] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[44] O. Maffeïs. *Ordonnancements de graphes de flots synchrones ; Application à la mise en œuvre de Signal*. PhD thesis, Université of Rennes, 1993.

[45] L. de Alfaro and T. Henzinger. Interface theories for component-based design. *Lecture Notes in Computer Science*, 2211:148–165, January 2001.

[46] F. Maraninchi and Y. Rémond. Mode-automata: about modes and states for reactive systems. In *ESOP '98: Proceedings of the 7th European Symposium on Programming*, pages 185–199. Springer-Verlag, 1998.

[47] J.T. Buck, S. Ha, E.A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *Int. Journal of Computer Simulation, special issue on "Simulation Software Development*, volume 4, pages 155–182, April 1994.

[48] F. Maraninchi and N. Halbwachs. Compiling argos into boolean equations. In *Lecture Notes in Computer Science*, volume 1135, pages 72–90, September 1996.

[49] J. R. Beauvais, R. Houdebine, P. Le Guernic, E. Rutten, and I. Gautier. A translation of statecharts and activitycharts into Signal equations. Technical Report Research Report No. 3397, INRIA, July 1999.

[50] J-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous dataflow with state machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, September 2005.

[51] O. Maffeïs and P.p Le Guernic. Distributed implementation of Signal: Scheduling & graph clustering. In *FTRTFT*, pages 547–566, 1994.

[52] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering*, 25(3):416–427, 1999.

[53] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/Lustre to TTA: A layered approach for distributed embedded applications. In *Proceedings of the LCTES 2003*, San Diego, Ca, US, June 2003.

[54] N. Pernet and Y. Sorel. Transformations de spécifications incluant du contrôle en spécification flot de données pour implantation distribuée. *Journal Européen des Systèmes Automatisés : Modélisation des Systèmes Réactifs (MSR'05)*, 39/1-3:31–46, 2005.

[55] N. Pernet and Y. Sorel. A design method for implementing specifications including control in distributed embedded systems. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'05)*, Catania, Italy, September 2005.

[56] J.M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P.A. Wacrenier. Data decision diagram for petri net analysis. In *Proceedings of the 23rd International Conference, ICATPN 2002, Lecture Notes in Computer Science*, Adelaide, Australia, June 2002.

[57] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proc. of 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, Ithaca, NY, 1987.

[58] M. von der Beeck. A comparison of statecharts variants. In *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems*, volume LNCS 863, pages 128–148. Springer-Verlag, 1994.

[59] C. André. Syncharts: a visual representation of reactive behaviors. Technical Report RR-95-52, I3S, October 1995.

[60] C. André. *The SyncCharts software environment: comprehensive examples of use*, september 1992.

[61] G. Berry. The constructive semantics of pure Esterel. (revision 1999).

[62] N. Pernet and Y. Sorel. Optimized implementation of distributed real-time embedded systems mixing control and data processing. In *Proceedings of the ISCA 16th International Conference: Computer Applications in Industry and Engineering (CAINE-2003)*, Las Vegas, Nv, US, November 2003.

[63] R. Nikoukhah and S. Steer. Scicos : A dynamic system builder and simulator, user's guide - version 1.0. Rapport de recherche INRIA 0207, INRIA, juin 1997.

[64] R. Djenidi, R. Nikoukhah, S. Steer, and Y. Sorel. Interface Scicos-SynDEx. Rapport de recherche 4250, INRIA, September 2001.

[65] Q. Pan, T. Gautier, L. Besnard, and Y. Sorel. Signal to SynDEx: translation between synchronous formalisms. disponible sur www.syndex.org/pub.htm, non-publié, 2003.

[66] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SynDEx software environment for real-time distributed systems, design and implementation. In *Proceedings of European Control Conference, ECC'91*, Grenoble, France, July 1991.

[67] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[68] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[69] C. Lavarenne, Claude Milan, Michel Paindavoine, G. Richard, and Y. Sorel. Implantation d'algorithmes de traitement d'images sur une architecture multi-DSP avec l'environnement d'aide à l'implantation SynDEx. In *Actes du XIVème Colloque GRETSI*, Juan-Les-Pins, France, September 1993.

[70] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* PhD thesis, Université de Paris Sud, Spécialité électronique, 30/11/2000.

[71] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2):103–112, 1991.

[72] T. Grandpierre and Y. Sorel. Un nouveau modèle générique d'architecture hététogène pour la méthodologie AAA. In *Actes des Journées Francophones sur l'Adéquation Algorithme Architecture, JFAAA'02*, Monastir, Tunisia, December 2002.

[73] T. Grandpierre and Y. Sorel. From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations. In *Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'03*, Mont Saint-Michel, France, June 2003.

[74] R. Seindal. *GNU M4, a powerful macro processor, version 1.4.4*, October 2005.

[75] T. Grandpierre, C. Lavarenne, and Y. Sorel. Modèle d'exécutif distribué temps réel pour SynDEx. Rapport de Recherche 3476, INRIA, August 1998.

[76] L. Kaouane, M. Akil, T. Grandpierre, and Y. Sorel. A methodology to implement real-time applications onto reconfigurable circuits. *Journal of Supercomputing*, 30(3):362–376, December 2004.

[77] L. Kaouane, M. Akil, Y. Sorel, and T. Grandpierre. From algorithm graph specification to automatic synthesis of fpga circuit: a seamless flow of graph transformations. In *Proceedings of 13th international conference on Field-Programmable Logic and Applications, FPL'03*, Lisbon, Portugal, September 2003.

[78] L. Kaouane, M. Akil, Y. Sorel, and T. Grandpierre. A methodology to implement real-time applications on reconfigurable circuits. In *Proceedings of International Conference on*

Engineering of Reconfigurable Systems and Algorithms, ERSA'03, Las Vegas, USA, June 2003.

[79] L. Kaouane. *Formalisation et optimisation d'applications s'exécutant sur architecture reconfigurable*. PhD thesis, Université de Marne-La-Vallée, Spécialité Informatique, 17/12/2004.

[80] A. Girault, H. Kalla, and Y. Sorel. An active replication scheme that tolerates failures in distributed embedded real-time systems. In *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'04*, Toulouse, France, August 2004. Kluwer Academic.

[81] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant static schedules. In *Proceedings of International Conference on Dependable Systems and Networks, DSN'03*, San Francisco, California, USA, June 2003.

[82] H. Kalla. *Génération automatique de distributions/ordonnancements temps réel fiables et tolérant les fautes*. PhD thesis, Institut National Polytechnique de Grenoble, Spécialité Systèmes et Logiciel, 17/12/2004.

[83] M. Raulet, F. Urban, J.-F. Nezan, C. Moy, O. Déforges, and Y. Sorel. Rapid Prototyping For Heterogeneous Multicomponent Systems: An MPEG-4 Stream Over An UMTS Communication Link. *Journal of Applied Signal Processing (JASP)*, 2005.

[84] M. Raulet, M. Babel, J.-F. Nezan, O. Déforges, and Y. Sorel. Automatic coarse-grain partitioning and automatic code generation for heterogeneous architectures. In *Proceedings of IEEE Workshop on Signal Processing Systems, SiPS'03*, Seoul, Korea, August 2003.

[85] R. Kocik and Y. Sorel. A methodology to design and prototype optimized embedded robotic systems. In *Proceedings of 2nd IMACS International Multiconference, CESA'98*, Hammamet, Tunisia, April 1998.

[86] R. Kocik and Y. Sorel. A methodology to reduce the design lifecycle of real-time embedded control systems. In *Proceedings of European Simulation and Modelling Conference, ESM'04*, Paris, France, October 2004.

[87] K. Ramamritham and J. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Software*, july 1984.

[88] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Management science research project 43, University of California, Los Angeles, USA, 1955.

[89] W.A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[90] M. Spuri, G.C. Buttazzo, and F. Sensini. Robust aperiodic scheduling under dynamic priority systems. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, pages 210–221, 1995.

[91] A.K. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 22, Washington, DC, USA, 1996. IEEE Computer Society.

[92] K. Jeffay, D.F. Stanat, and C.U. Martel. On non-preemtive scheduling of periodic and sporadic tasks. *IEEE*, 1991.

[93] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1), 1989.

[94] M. Spuri and G.C. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2), 1996.

[95] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G.C. Buttazzo, M. Caccamo, J. Lehoczky, and A.K. Mok. Real time scheduling theory : A historical perspective. *Real-Time Systems*, 28:101–155, 2004.

[96] J. Leung and M. Merrill. A note on preemptive scheduling of periodic real-time tasks. *IEEE Informations Processing Letters*, 11(3):115–118, 1980.

[97] J. Leung and J.W. Whitehead. On the complexity of fixed priority scheduling of periodic real-time tasks. *Performance Evalutation*, 2(4):237–250, 1982.

[98] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 1973.

[99] J.P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic sheduling algorithm: exact characterization and average case bahavior. *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[100] E. Bini, G.C. Buttazzo, and G. M. Buttazzo. Rate monotonic scheduling: The hyperbolic bound. *IEEE Transactions on Computers*, 52(7):933–942, jully 2003.

[101] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Hard real-time scheduling : The deadline-monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, GA, USA, 1991.

[102] A.K. Mok. Fundamental design problems of distributed systems for the hard real-time environment. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[103] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2(4):301–324, 1990.

[104] G.C. Buttazzo. Rate monotonic vs. EDF: Judgment day. In *Proceedings of the 3rd ACM International Conference on Embedded Software (EMSOFT 2003)*, Philadelphia, October 2003.

[105] H. Chetto, Silly M., and Bouchentouf T. Dynamic scheduling of real-time tasks under precedence contraints. *The Journal of Real-Time Systems*, 1990.

[106] T. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Transaction on ComputersReal-Time Systems*, 53(3):334–350, 2004.

[107] S.K. Baruah. A general model for recurring real-time tasks. In *Real-Time Systems Symposium*, pages 114–122, Madrid, Spain, December 1998.

[108] S.K. Baruah, D. Chen, S. Gorinsky, and A.K. Mok. Generalized multiframe tasks. *The International Journal of Time-Critical Computing Systems*, 17:5–22, 1999.

[109] B. Wittenmark, J. Nilsson, and M. Torngren. Timing problems in real-time control systems. In *Proceedings of the American Control Conference*, Seattle, Washington, 1995.

[110] L. Sha and J.B. Goodenough. Real-time scheduling theory and ada. *IEEE Computer*, 23(4):53–62, 1990.

[111] R. Dobrin and G. Fohler. Attribute assignment for the integration of off-line and fixed priority scheduling. In *Work-in-Progress Session, 21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, Florida, USA, November 2000.

[112] G. Fohler, T. Lennvall, and G.C. Buttazzo. Improved handling of soft aperiodic tasks in offline scheduled real-time systems using total bandwidth server. In *8th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, Nice, France, October 2001.

[113] J.P. Lehoczky, L. Sha, and J.K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-time Symposium*, December 1987.

[114] J.K. Strosnider, J.P. Lehoczky, and L. Sha. The deferrable server algorithm for enhancing aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, january 1995.

[115] J.P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. *Real-Time Systems Symposium*, 1992.

[116] R.I. Davis, K.W. Tindell, and A. Burns. Scheduling slack time in fixed priority preemptive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.

[117] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings. Real-Time Systems Symposium*, 1988.

[118] S. Ramos-Thuel and J. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.

[119] J.P. Lehoczky and S. Ramos-Thuel. Scheduling periodic and aperiodic tasks using the slack stealing algorithm, 1995.

[120] T.M. Ghazalie and T.P. Baker. Aperiodic servers in a deadline scheduling environment. *The Journal of Real-time Systems*, 1995.

[121] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 1989.

[122] H. Chetto and M. Chetto. Scheduling periodic and sporadic tasks in a real-time system. *Information Processing Letters*, 30, Issue 4:177–184, February 1989.

[123] K. Schwan and H. Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 8:736–748, 1992.

[124] G.C. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Trans. Comput.*, 48(10):1035–1052, 1999.

[125] G. Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *IEEE Real-Time Systems Symposium*, pages 152–161, 1995.

[126] D. Isovic and G. Fohler. Handling sporadic tasks in off-line scheduled real-time systems. In *11th EUROMICRO Conference on Real-Time Systems*, York, UK, July 1999.

[127] D. Isovic and G. Fohler. Online handling of firm aperiodic tasks in time triggered systems. In *12th EUROMICRO Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[128] D. Isovic and G. Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *21st IEEE Real-Time Systems Symposium*, Walt Disney World, Orlando, Florida, USA, November 2000.

[129] A. Cervin and J. Eker. Feedback scheduling of control tasks. In *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[130] K.-E. Årzén, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Proceedings of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, December 2000.

[131] C. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. A feedback control approach for guaranteeing relative delays in web servers. In *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*, page 51, Washington, DC, USA, 2001. IEEE Computer Society.

[132] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, and S. Son. Feedback control scheduling in distributed real-time systems. In *Real-Time Systems Symposium*, 2001.

[133] T.-S. Tia, J.-W.-S. Liu, and M. Shankar. Algorithms and optimality of scheduling soft aperiodic requests in fixed-priority preemptive systems. *Journal of Real-Time Systems*, 10(1), 1996.

[134] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *20th IEEE Real-Time Systems Symposium (RTSS'99)*, Phoenix, USA, December 1999.

[135] M. Torngren. Fundamentals of implementing real-time control application in distributed computer systems. In *Journal of Real-Time Systems*, number 14, pages 219–260, 1998.

[136] M. Klein, J.P. Lehoczky, and R. Rajkumar. Rate-monotonic analysis for real-time industrial computing. In *IEEE Computer*, pages 24–33, 1994.

[137] L. Cucu. *Ordonnancement non préemptif et condition d'ordonnançabilité pour systèmes embarqués à contraintes temps réel*. PhD thesis, Université Paris-Sud XI, France, 2004.

[138] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. *Real-time and Embedded Systems*, 2002.

[139] R. Bettati and J.W.S. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1990.

[140] R. Gerber, S. Hong, and M. Saksena. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Real-Time Systems Symposium*, San Juan, Puerto Rico, 1994.

[141] L. Cucu and Y. Sorel. Non-preemptive scheduling algorithms and schedulability conditions for real-time systems with precedence and latency constraints. Research report RR-5403, INRIA, Rocquencourt, France, 2004.

[142] R. Dobrin, G. Fohler, and P. Puschner. Translating off-line schedules into task attributes for fixed priority scheduling. In *22nd IEEE Real-Time Systems Symposium*, London, UK, October 2001.

[143] M.R. Garey, D.S. Johnson, B.B. Simons, and R.E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deaedlines. *IEEE Transactions on Software Engineering*, 10(2):256–269, May 1981.

[144] H. Chetto and M. Chetto. How to insure feasibility in distributed system for real-time control. In *International Symposium on High Performance Comuter Systems*, pages 98–105, Paris, France, December 1987.

Index

activation (tâche), 136
active ou prête (tâche), 136
algorithme, 25
algorithme à priorités dynamiques, 141
algorithme à priorités fixes, 141
arc d'activation, 77
arc instantané (SyncCharts), 57
arc régulier, 77
Argos, 28
atomique, 25

capacité (d'un serveur), 151
chevauchement (contraintes de latence), 184
communication conditionnée, 43
CondI (opération), 44
condition (d'une opération), 40
condition exclusive, 41
CondO (opération), 49
constellation (SyncCharts), 55
contrainte temps réel, 136
contrôle, 26

date d'activation, 138
date de début au plus tôt, 182
date de début d'exécution, 139
date de fin d'exécution, 139
Deadline Monotonic, 142
dépendance conditionnée, 41
dépendance conditionnée entrante, 43
dépendance conditionnée interne, 43
dépendance conditionnée sortante, 43
dépendance de conditionnement, 40
dépendance de données, 28
donnée de conditionnement, 40
durée d'exécution au pire cas, 138
durée minimum d'inter-activations, 137

Earliest Deadline First, 143
échéance, 136
échéance absolue, 138
échéance relative, 138
en-ligne, 147
Esterel, 28
état précédent, 58
état courant, 58
étoile (SyncCharts), 55
exclusion mutuelle, 41

facteur d'utilisation processeur, 142
feedback scheduling, 167
flot de contrôle, 27
flot de données, 28
FSM (Finite State Machine), 27

graphe de précédence, 139

hétérogène (flot de données), 29
homogène (flot de données), 29
hors-ligne, 147
hypergraphe orienté acyclique d'activation (HOAA), 91
hyperpériode, 141

inactive (tâche), 136
inclusion (contraintes de latence), 184
instance (d'une tâche), 138
instant logique, 26
intervalle d'exécution, 191

jeu de tâches, 139

langage de programmation, 26
langage synchrone, 26
latence, 180

laxité, 143
Least Laxity First, 143
Lustre, 30

machine à états finis, 27
machine de Von Neumann, 25
macro-état (SyncCharts), 55
modèle de programmation, 26

non préemptif, 140

opération, 28
opération conditionnante, 40
opération conditionnée, 40
optimalité pour ordonnancement de tâches périodiques, 140
optimalité pour ordonnancement de tâches périodiques et tâches apériodiques fermes, 149, 200
optimalité pour ordonnancement de tâches périodiques et apériodiques souples, 149
ordonnançabilité, 140
ordonnançable, 140
ordonnancement, 140
ordonnancement correct ou valide, 140
ordonnancement en tâche de fond, 149
ordonnanceur, 147

parallélisme potentiel, 30
période, 136, 138
phase, 138
pire temps de réponse, 142
précédence, 139
prédécesseur, 29, 140
prédécesseur direct, 29, 140
préemptif, 140
préemption, 140
préemption faible (SyncCharts), 57
préemption forte (SyncCharts), 56
programme, 25

Rate Monotonic, 141
réactif (système), 15
règle d'activation (flot de données), 29
réincarnation, 68
rémanence, 53
retard, 29

Scicos, 76
serveur ajournable, 152
serveur ajournable dynamique, 158
serveur à échange de priorités, 154
serveur à échange de priorités dynamique, 159
serveur à échange de priorités dynamique amélioré, 165
serveur à scrutation, 151
serveur à scrutation dynamique, 158
serveur à utilisation totale, 162
serveur à utilisation totale (version améliorée), 166
serveur à utilisation totale (version optimale), 165
serveur Earliest Deadline Late, 163
serveur sporadique, 155
serveur sporadique dynamique, 160
Signal, 30
signal d'activation, 77
signal régulier, 77
slack stealing, 157
slot, 190
slot shifting, 166, 190
spare capacity, 166, 192
Statecharts, 28
successeur, 29, 140
successeur direct, 29, 140
suspension (SyncCharts), 57
SyncCharts, 28, 55
synchrone (langage), 26
synchrone (potentiellement), 81
SynDEx (langage), 52
SynDEx (logiciel), 103
système, 15

tâche, 135
tâche apériodique, 136
tâche apériodique ferme, 137, 139
tâche apériodique souple, 137, 139

tâche d’entrée, 140
tâche de sortie, 140
tâche périodique, 136
tâche sporadique, 137
temps réel souple (ou mou), 136
temps réel strict, 136
terminaison normale (SyncCharts), 57
test d’acceptation, 137

uniformisation des échéances, 183

Implantation distribuée temps réel de programmes conditionnés à l'aide d'ordonnancements mixtes hors-ligne en-ligne de tâches périodiques avec contraintes de latence et acceptation de tâches apériodiques

Résumé

Cette thèse traite de deux aspects importants de l'implantation distribuée des systèmes temps réel. Le premier concerne l'implantation distribuée automatique de programmes conditionnés. Ainsi, nous proposons un modèle flot de données conditionné bien adapté pour implantation distribuée et montrons comment des programmes conditionnés peuvent être traduits automatiquement en programmes utilisant ce modèle avant d'être ordonnancés et distribués efficacement à l'aide du logiciel SynDEx. Le deuxième aspect concerne l'exécution conjointe de tâches apériodiques et périodiques. Les premières s'exécutent selon un ordonnancement effectué en-ligne alors que les secondes, issues de programmes conditionnés, s'exécutent selon un ordonnancement calculé hors-ligne. Nous proposons deux extensions de la méthode de slot shifting qui permettent dans ce contexte de respecter des contraintes de latence. Cette contrainte permet de définir la date limite de fin d'exécution d'une tâche par rapport au début de l'exécution d'une autre. Nous comparons les deux extensions proposées en termes de complexité et d'optimalité.

Mots-clés : temps réel distribué, conditionnement, latence, tâches périodiques et aperiodiques.

Distributed real-time implementation of conditionned programs with off-line and on-line scheduling of periodic tasks with latency constraints and aperiodic tasks acceptance

Abstract

This thesis focuses on two important aspects of the implementation of distributed real-time systems. The first one concerns the automatized distributed implementation of conditionned programs. We introduce a new conditionned data flow model which is well-suited for distributed implementation and we show how to automatically translate conditionned programs into programs based on this model. Thus, we describe how the SynDEx software allows to efficiently schedule and distribute the obtained programs. The second aspect concerns the combined execution of aperiodic and periodic tasks. The first ones are executed according to an on-line schedule contrary to the second which are executed according to an off-line schedule. In this context we propose two extensions of the slot shifting method which allows to satisfy latency constraints. A latency constraint defines a latest finishing time of a task relatively to the start time of another one. We compare these two extensions in terms of complexity and optimality.

Key-words: distributed real-time, conditioning, latency, periodic and aperiodic tasks.