INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
/ / / / / / / / / / /

THESE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : Informatique « Systèmes et Logiciels »

préparée au laboratoire : INRIA Rhône-Alpes
dans le cadre de l'**École Doctorale « Mathématiques, Sciences et technologies de l'information, Informatique »**

présentée et soutenue publiquement
par

Hamoudi KALLA

le 17 décembre 2004

Titre :

Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes

Directeur de thèse : Alain GIRAULT

JURY

M.	Denis TRYSTRAM	Président
M.	Yves CROUZET	Rapporteur
Mme	Pascale MINET	Rapportrice
Mme	Isabelle PUAUT	Examinatrice
M.	Alain GIRAULT	Directeur de thèse
M.	Yves SOREL	Co-directeur de thèse

« La reconnaissance silencieuse ne sert à personne. »
GLADYS BRONWYN STERN

Remerciements

Je remercie vivement Alain Girault, mon directeur de thèse, pour m'avoir encadré pendant toutes ces années, cette thèse lui doit beaucoup. La disponibilité, la confiance et la liberté qu'il m'a accordées au cours de ce travail de thèse, m'ont permis d'entreprendre de nombreuses expériences et ont grandement contribué à la richesse de cette thèse. Je le remercie pour tous les matches de foot que nous avons joué ensemble, sans oublier qu'il soutient le . . . , et que je soutiens l'OM.

Je remercie également Yves Sorel pour avoir co-encadré cette thèse. Sa patience, sa disponibilité et ses commentaires m'ont permis d'accomplir ce travail dans les meilleures conditions qui soient. Je remercie tous les membres de son équipe pour leurs commentaires, ainsi leur accueil à l'INRIA ROCQUENCOURT.

J'exprime ma sincère reconnaissance aux membres du jury, Denis Trystram, Président, Pascale Minet et Yves Crouzet, Rapporteurs, Isabelle Puaut, Examinatrice. Leurs commentaires et les conversations que j'ai pu avoir avec eux ont largement contribués à l'amélioration de ce document.

J'aimerais remercier Bernard Espiau, directeur de l'INRIA RHÔNES-ALPES, et Eric Rutten, Responsable de l'équipe POP ART, de m'avoir hébergé à l'INRIA, où j'ai pu bénéficier de moyens exceptionnels.

Je voudrais aussi remercier tous les collègues passés et présents des projets POP ART et BIPOP qui par leurs conseils et leurs encouragements ont contribué à l'aboutissement de cette thèse.

Je réserve un remerciement chaleureux à mes chers amis : D. Noureddine *"oui, c'est grâce à toi"*, Z. Youcef *"toujours dans le coeur"*, C. Mohiedine *"j'adore ses soirées parisiennes"*, H. Riadh *"prof, lah ybarek"*, C. Sophie *"une amie adorable et pleine de gentillesse"*, B. Jean-Matthieu *"on peut toujours compter sur lui, merci Mamie ! !"*, G. Matthieu *"un ami au grand coeur :-)"*, T. Elodie *"toujours souriante"*, D. Emil *"mon 1er choco-loc de bureau"*, R. David *"mon 2eme cool-oc"*, D. Gwenaël *"mon 3eme nanocoloc, au petit estomac :-)"*, J. Sébastien *"PSG dima !"*, P. Valier *"père moderne d'Emma"*, M. Kamel *"et hop, tkakitlak !"*, L. Oussama, M. Jérôme, M. Marie, L. C. Gwen, L. Blandine, et C. Jean-Baptiste.

Je dois un grand merci à tous mes amis du Village Olympique : D. Hakim, F. Kamel, D. Sofiane, A. Khireddine, et N. Wahab. Merci pour tout, merci beaucoup, infiniment merci, merci encore...

Je réserve une reconnaissance particulière à B. A. Zohra *"coucou, j'adore tes gateaux ! ! ! ahhh :-)"*, M. Feryal Kamila *"une amie comme elle, il n'y en pas beaucoup !"*, et B. Fethi *"et voila, un vrai chaoui ! ! !"*.

Mes derniers remerciements s'adressent à ma famille. Je remercie tout particulièrement mes cinq soeurs, mes six frères, mon cousin Toufik, mes parents, qui m'ont toujours aidé, soutenu et encouragé au cours de mes études.

A mes parents adorables :-)
Amor et Fatima,

A ma chère amie !
Zahra,

Table des matières

Table des figures **7**

Liste des tableaux **9**

1 Introduction générale **11**
1.1 Problématique de la thèse . . . 12
1.2 Motivations et contributions . . . 13
1.3 Structure du mémoire . . . 14

I Concepts de base et terminologies 17

2 Introduction à la distribution et à l'ordonnancement temps réel **19**
2.1 Introduction . . . 19
2.2 Définitions . . . 20
2.2.1 Système réactif . . . 20
2.2.2 Système distribué et système embarqué . . . 21
2.3 Spécification des systèmes distribués réactifs embarqués . . . 22
2.3.1 L'algorithme . . . 22
2.3.2 L'architecture matérielle . . . 23
2.3.3 Les contraintes temporelles et matérielles . . . 23
2.4 Problème de distribution et d'ordonnancement temps réel . . . 24
2.4.1 Terminologies . . . 24
2.4.2 Présentation du problème . . . 25
2.5 Algorithmes de distribution et d'ordonnancement temps réel . . . 26
2.5.1 Algorithmes hors-ligne et en-ligne . . . 26
2.5.2 Algorithmes exacts et approchés . . . 27
2.6 Algorithme de distribution et d'ordonnancement de SYNDEX . . . 27
2.6.1 L'algorithme, l'architecture et les contraintes temporelles et matérielles . . . 27
2.6.2 Présentation de l'algorithme de distribution et d'ordonnancement . . . 28
2.7 Conclusion . . . 31

3 Tolérance aux fautes et fiabilité des systèmes réactifs 33
3.1 Introduction 33
3.2 Tolérance aux fautes des systèmes réactifs 34
3.2.1 Terminologies 34
3.2.2 Algorithmes de tolérance aux fautes 36
3.2.3 Problème de distribution et d'ordonnancement temps réel et tolérant aux fautes 39
3.3 Fiabilité des systèmes réactifs 40
3.3.1 Terminologies 40
3.3.2 Modèles pour le calcul de la fiabilité 41
3.3.3 Problème de distribution et d'ordonnancement temps réel et fiable 43
3.4 Conclusion 44

II Méthodologies pour la génération de distributions et d'ordonnancements temps réel, fiables et tolérants aux fautes 45

4 Modèles 47
4.1 Introduction 47
4.2 Modèle d'algorithme 48
4.2.1 Hypergraphe orienté 48
4.2.2 Hypergraphe orienté infiniment répété 49
4.3 Modèle d'architecture 52
4.3.1 Architecture à liaisons point-à-point 56
4.3.2 Architecture à liaisons bus 56
4.4 Modèle d'exécution 57
4.4.1 Exécution cyclique de l'algorithme sur l'architecture 58
4.4.2 Contraintes liées à l'architecture et au temps réel 60

5 État de l'art 63
5.1 Introduction 63
5.2 Algorithmes de distribution et d'ordonnancement temps réel et tolérants aux fautes 64
5.2.1 Algorithmes basés sur la redondance active 64
5.2.2 Algorithmes basés sur la redondance passive 66
5.2.3 Algorithmes basés sur la redondance hybride 69
5.2.4 Comparaison 70
5.3 Algorithmes de distribution et d'ordonnancement temps réel et fiables 71
5.3.1 Algorithmes uni-objectif 71
5.3.2 Algorithmes multi-objectifs 71
5.3.3 Discussion 72
5.4 Conclusion 73

6 Méthodologie AAA-TP pour des architectures à liaisons point-à-point 75
6.1 Présentation du problème bi-objectifs de tolérance aux fautes et de prédictibilité . . 75
6.1.1 Modèle de fautes 76
6.1.2 Données du problème 77
6.2 Principe général de la méthodologie AAA-TP 78
6.3 Phase d'initialisation : transformation de graphe 80
6.3.1 Notations et définitions 80
6.3.2 Tolérance aux fautes des processeurs 83
6.3.3 Tolérance aux fautes des liens de communication 84
6.3.4 Tolérance aux fautes des processeurs et des liens de communication 85
6.4 Phase d'adéquation : heuristique de distribution et d'ordonnancement 90
6.4.1 Notations 91
6.4.2 Principes de l'heuristique 91
6.4.3 Présentation de l'heuristique 93
6.5 Prédiction du comportement temps réel 97
6.6 Extension de la méthodologie AAA-TP à la tolérance aux fautes des capteurs . . . 98
6.6.1 Complexité du problème 99
6.7 Simulations 100
6.7.1 Les paramètres de simulation 100
6.7.2 Les résultats 100
6.8 Conclusion 102

7 Méthodologie AAA-TB pour des architectures à liaisons bus 103
7.1 Présentation du problème de tolérance aux fautes 103
7.1.1 Modèle de fautes 104
7.2 Principe général de la méthodologie AAA-TB 105
7.2.1 Redondance active des opérations et passive des communications 106
7.2.2 Fragmentation des données de communication 106
7.2.3 Tolérance aux fautes des processeurs 107
7.2.4 Tolérance aux fautes des bus de communication 110
7.2.5 Tolérance aux fautes des processeurs et des bus de communication 110
7.3 Heuristique de distribution et d'ordonnancement 115
7.3.1 Présentation de l'heuristique 115
7.4 Simulations 117
7.4.1 Les paramètres de simulation 117
7.4.2 Les résultats 117
7.5 Conclusion 119

8 Méthodologie AAA-F 121
8.1 Présentation du problème bi-critères de temps réel et de fiabilité 121
8.1.1 Modèle de fautes 122
8.1.2 Données du problème 122

8.2 Principe général de la méthodologie AAA-F 123
8.2.1 Calcul de la longueur d'une distribution/ordonnancement 125
8.2.2 Calcul de la fiabilité d'une distribution/ordonnancement 125
8.2.3 Deux critères antagonistes 133
8.3 Heuristique de distribution et d'ordonnancement bi-critères 133
8.3.1 Principe de l'heuristique 133
8.3.2 Présentation de l'algorithme de l'heuristique 136
8.4 Simulations 140
8.4.1 Les paramètres de simulation 140
8.4.2 Les résultats 140
8.5 Conclusion 141

III Développement logiciel 143

9 Le logiciel SYNDEX 145
9.1 Présentation générale 145
9.2 Spécification 145
9.2.1 Graphe d'algorithme 145
9.2.2 Graphe d'architecture 147
9.2.3 Caractéristiques d'exécution 147
9.3 Heuristiques de distribution/ordonnancement temps réel 147
9.3.1 Heuristique uni-critère de la méthodologie AAA 148
9.3.2 Heuristique tolérante aux fautes de la méthodologie AAA-TP 148
9.3.3 Heuristique tolérante aux fautes de la méthodologie AAA-TB 149
9.3.4 Heuristique bi-critères de la méthodologie AAA-F 150
9.4 Génération automatique d'exécutifs 150

10 Génération aléatoire de graphes d'algorithmes et d'architectures 153
10.1 Générateur de graphe d'algorithme 154
10.2 Générateur de graphe d'architecture 155
10.3 Caractérisation de graphe d'algorithme et d'architecture 155

Conclusion et perspectives 157

Bibliographie 160

Table des figures

2.1 Modèle d'un système réactif. 20
2.2 Exemple d'un système réactif. 21
2.3 Exemple d'un algorithme. 23
2.4 Exemple d'une architecture distribuée. 24

3.1 Relation entre faute, erreur et défaillance 35
3.2 Couverture entre les hypothèses de défaillance 37

4.1 Exemple d'un graphe d'algorithme. 49
4.2 Exemple d'un graphe d'algorithme infiniment répété. 50
4.3 Exemple d'un graphe d'algorithme factorisé. 51
4.4 Exemple d'une architecture distribuée. 53
4.5 Exemple d'une architecture distribuée à liaisons point-à-point. 54
4.6 Exemple d'une architecture distribuée à liaisons bus. 55
4.7 Modèles classiques. 55
4.8 Modèle d'une architecture à liaisons point-à-point. 56
4.9 Modèle d'une architecture à liaisons bus. 57

5.1 Exemple de la redondance active. 65
5.2 Tolérance aux fautes des processeurs. 67
5.3 Tolérance aux fautes des média de communication. 68
5.4 La redondance hybride. 69

6.1 Méthodologie AAA-TP. 80
6.2 Forme d'une opération de contrôle. 81
6.3 Exemple de transformation d'un graphe d'algorithme. 82
6.4 Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 0$. 83
6.5 Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf \geq 1$ et $\mathcal{N}lf = 0$. 84
6.6 Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf = 0$ et $\mathcal{N}lf \geq 1$. 85
6.7 Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf \geq 0$ et $\mathcal{N}lf \geq 0$. 86
6.8 Schéma de transformation de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 1$. 87
6.9 Schéma de transformation final de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 1$. 88
6.10 Schémas de transformation réduits. 88
6.11 Réplication de certaines opérations en plus de $\mathcal{N}pf$+1 copies. 89

6.12 Schéma de transformation final de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf \geq 1$ et $\mathcal{N}lf \geq 1$. 90
6.13 Exemple d'une distribution/ordonnancement. 92
6.14 Modèle d'une architecture avec deux capteurs. 98
6.15 Effet de la réplication active pour $\mathcal{N}pf = 1$, $P = 6$ et $N = 50$. 101
6.16 Effet de N sur AAA-TP et HBP pour $\mathcal{N}pf = 1$, $P = 4$ et $CCR = 5$. 101
6.17 Effet de CCR sur AAA-TP et HBP pour $\mathcal{N}pf$=1, P=4 et N=50. 102

7.1 Défaillance d'un bus de communication 104
7.2 Méthodologie AAA-TB. 105
7.3 Redondance active des opérations et passive des communications. 106
7.4 Fragmentation des données de communications. 107
7.5 Tolérance aux fautes des processeurs. 108
7.6 Algorithme de fragmentation de données. 109
7.7 Recouvrement rapide des erreurs. 109
7.8 Tolérance aux fautes des bus de communication. 110
7.9 Tolérance aux fautes des processeurs et des bus de communication. 111
7.10 Tolérance aux fautes des processeurs. 112
7.11 Tolérance aux fautes des bus de communication. 113
7.12 Exemple d'application de la méthodologie AAA-TB. 114
7.13 Effet de N et $\mathcal{N}bf$ pour $\mathcal{N}pf = 0$, $P = 5$ et $CCR = 1$. 118
7.14 Effet de CCR pour $\mathcal{N}pf = 1$, $\mathcal{N}bf = 2$, $P = 5$ et $N = 40$. 118

8.1 Méthodologie AAA-F. 124
8.2 Exemple d'un système sans redondance en série. 126
8.3 Exemple d'un système avec redondance en parallèle. 128
8.4 Exemple d'un système en série/parallèle sans fautes de liens de communication. . . 129
8.5 Exemple d'un système en structure quelconque 130
8.6 Transformation de graphe d'algorithme. 131
8.7 Nouveau système en série/parallèle après transformation du graphe d'algorithme. . 132
8.8 Calcul du gain en longueur et de la perte en fiabilité. 134
8.9 Calcul d'un compromis entre $\mathcal{G}^{(n)}$ et $\mathcal{P}^{(n)}$. 136
8.10 Comparaison en longueur de la distribution/ordonnancement. 140
8.11 Comparaison en fiabilité de la distribution/ordonnancement. 141

9.1 Le logiciel SynDEx. 146
9.2 Exemple d'un graphe d'algorithme $\mathcal{A}lg$. 146
9.3 Exemple d'un graphe d'architecture $\mathcal{A}rc$. 147
9.4 Spécification des coûts d'exécution $\mathcal{E}xe$. 148
9.5 Distribution/ordonnancement générée par l'heuristique de AAA. 148
9.6 Distribution/ordonnancement générée par l'heuristique de AAA-TP. 149
9.7 Distribution/ordonnancement générée par l'heuristique de AAA-TB. 149
9.8 Distribution/ordonnancement générée par l'heuristique de AAA-F. 150
9.9 Exemple d'un macro-exécutif du processeur P1. 151

9.10 Processus de génération d'un exécutif par SYNDEX. 152

10.1 Processus de génération aléatoire de graphes. 153
10.2 Étapes de génération aléatoire d'un graphe d'algorithme. 154

Liste des tableaux

4.1 Durées d'exécution $\mathcal{E}xe$ pour les opérations. 59
4.2 Durées de transfert $\mathcal{E}xe$ pour les dépendances de données. 60

5.1 Comparaison entre les trois approches de redondance. 70

« Il nous faut peu de mots pour exprimer l'essentiel ;
il nous faut tous les mots pour le rendre réel. »
Paul ELUARD

Chapitre 1

Introduction générale

D'année en année, les systèmes réactifs envahissent de nombreux secteurs d'activité dans notre vie quotidienne. Le contrôle d'une centrale nucléaire, les systèmes d'aide au pilotage des avions et les téléphones portables sont quelques exemples de systèmes réactifs. Les progrès accomplis en électronique et en informatique ont apporté beaucoup à l'augmentation de la puissance de calcul des machines et à l'amélioration des performances de ces systèmes.

Dans certains secteurs d'activité, tels que l'automobile, l'aéronautique, le contrôle/commande de processus industriels, l'énergie et le secteur militaire, les nouveaux systèmes réactifs sont de plus en plus petits et plus rapides, mais plus complexes et critiques, et donc plus sensibles aux *fautes*. La présence de certaines fautes, qu'elles soient accidentelles (conception, interaction, etc.) ou intentionnelles (humaines, virus, etc.), est inévitable. Concevoir un système réactif *idéal* sans fautes est donc une tâche difficile voire impossible à réaliser. Mais, au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une défaillance, ces systèmes doivent être sûrs de fonctionnement.

Notre travail se place dans le cadre général de la conception des systèmes réactifs sûrs de fonctionnement. Plus particulièrement, nous nous intéressons à des systèmes réactifs distribués et embarqués. L'architecture matérielle d'un système réactif distribué est composée de plusieurs calculateurs, capteurs et actionneurs, lui apportant plus de puissance de calcul et de modularité. Les éléments de cette architecture distribuée sont reliés entre eux par un ensemble de média de communication, tels que des liens point-à-point, des bus et des mémoires partagées. Dans le cadre des systèmes réactifs distribués et embarqués, le nombre de ces éléments est réduit au strict minimum, autant pour des raisons physiques (dimensions, poids, consommation électrique, etc.) que pour des raisons de coûts de conception ou de production. Des exemples de ce type de systèmes sont le système s'occupant des freins anti-bloquants, le système de déclenchement des airbags, et le système d'injection de carburant.

La sûreté de fonctionnement de ces systèmes peut être obtenue par plusieurs méthodes [50], tels que des méthodes basées sur la *tolérance aux fautes* et sur une mesure de *fiabilité*, qui sont l'objet de cette thèse :

- La *tolérance aux fautes* permet à un système de continuer à délivrer un service conforme à sa spécification même en présence de défaillances. Elle peut être obtenue par l'utilisation de solutions logicielles, matérielles, ou une combinaison des deux. L'approche matérielle consiste à introduire un ensemble de redondances physiques dans le système, tels que des processeurs, des média de communication et des capteurs. L'approche logicielle consiste à introduire un ensemble de redondances logicielles dans le système, tels que des voteurs logiciels. À la différence des solutions matérielles, qui ne peuvent tolérer que des fautes physiques, les solutions logicielles peuvent être utilisées pour tolérer des fautes logicielles et matérielles. Le choix entre la redondance matérielle et logicielle dépend des caractéristiques du système à concevoir.
- La *fiabilité* est la mesure qui permet d'évaluer quantitativement la sûreté de fonctionnement d'un système. L'évaluation quantitative consiste à fournir une mesure probabiliste du bon fonctionnement d'un système durant son exploitation. Plusieurs méthodes ont été développées, dans la littérature, pour améliorer et estimer la fiabilité d'un système. L'amélioration de la fiabilité passe par l'utilisation de solutions de redondance matérielle ou logicielle, et l'estimation de la fiabilité peut être effectuée par plusieurs modèles de calcul, tels que les modèles combinatoires. Enfin, comme pour la tolérance aux fautes, le choix entre la redondance matérielle et logicielle dépend aussi des caractéristiques du système à concevoir.

1.1 Problématique de la thèse

La « croissance continue de la complexité des processeurs et l'arrivée de nouvelles technologies indiquent la persistance du problème des fautes des processeurs, qui peut devenir pire dans l'avenir [8] ». En plus des fautes des processeurs, les fautes des média de communication sont parmi les fautes les plus fréquentes dans un système distribué. C'est donc à la problématique de la conception des systèmes réactifs fiables et tolérants aux fautes matérielles des processeurs et des média de communication que cette thèse apporte une contribution. Pour résoudre ce problème de conception, plusieurs techniques ont été proposées dans la littérature. Elles sont basées sur l'utilisation de solutions logicielles, matérielles ou une combinaison des deux. Étant donné que nous visons des systèmes embarqués, il est impossible d'ajouter des processeurs redondants sans aller à l'encontre des contraintes de coût, de taille, de consommation électrique… Nous ne nous intéressons donc dans cette thèse qu'aux solutions logicielles, c'est-à-dire que nous utilisons au mieux la redondance matérielle existante dans l'architecture du système sans essayer de rajouter des ressources physiques supplémentaires. Plus particulièrement, nous cherchons à résoudre deux problèmes NP-difficiles :

- le problème de la distribution/ordonnancement temps réel, tolérante aux fautes et prédictive, et
- le problème de la distribution/ordonnancement temps réel, fiable et prédictive.

Notre premier objectif dans cette thèse consiste donc à proposer des techniques logicielles qui permettent, à partir d'une spécification d'un algorithme et d'une architecture, de fournir une distribution et un ordonnancement des composants logiciels de l'algorithme sur les composants matériels de l'architecture du système, tout en garantissant que les contraintes de distribution, de temps réel, et de *tolérance aux fautes* soient respectées. Les contraintes de distribution définissent une relation d'exclusion entre certains composants matériels et certains composants logiciels. La contrainte temps réel définit une borne maximale sur l'exécution de l'algorithme sur l'architecture. Les contraintes de tolérance aux fautes définissent des hypothèses sur le nombre maximal de fautes des processeurs et le nombre maximal de fautes des média de communication que le système doit tolérer. Concernant les média de communication, nous supposons qu'ils sont tous ou bien de type point-à-point, ou bien de type bus. Par conséquent le processus de distribution/ordonnancement de l'algorithme sur l'architecture devra prendre en compte cette différence afin d'exploiter les avantages existants au niveau des caractéristiques de chaque type de média.

Le deuxième objectif de cette thèse consiste à proposer des techniques logicielles qui permettent de fournir une distribution et un ordonnancement des composants logiciels de l'algorithme sur les composants matériels de l'architecture du système, tout en garantissant que les contraintes de distribution, de temps réel, et de fiabilité soient respectées. Les contraintes de distribution et de temps réel sont les mêmes contraintes que dans notre premier objectif. La contrainte de fiabilité définit une borne maximale sur la probabilité de défaillance du système, ou autrement dit que la probabilité d'exécution correcte du système doit être supérieure à un seuil de fiabilité.

Enfin, dans les deux cas, la distribution/ordonnancement doit être *prédictive*. Puisque toutes les caractéristiques des composants logiciels (exécution, distribution, etc.) de l'algorithme et des composants matériels (connexion, type de média, etc.) de l'architecture sont connues, prédictif signifie qu'il est possible de déterminer, *avant* la mise en exploitation du système, si oui ou non les contraintes temps réel sont respectées. Cette vérification peut même être raffinée pour tenir compte de la présence/absence de fautes à l'exécution.

1.2 Motivations et contributions

Les solutions que nous proposons dans cette thèse sont classées dans la catégorie des solutions logicielles et basées sur la redondance logicielle. Les deux principales motivations de ce choix sont :

- répondre aux contraintes d'embarquabilité imposées par les systèmes embarqués, telles que la consommation d'énergie, l'encombrement ...
- répondre aux attentes des industries d'aujourd'hui qui exigent de moindres coûts de conception de leur systèmes.

Afin de résoudre les deux problèmes de conception de systèmes fiables et tolérants aux fautes, nous proposons dans cette thèse trois méthodologies de conception :

1. La première méthodologie consiste à générer automatiquement des distributions/ordonnancements tolérantes aux fautes, adaptées aux architectures matérielles distribuées et hétérogènes munies d'un réseau de communication composé uniquement de liaisons point-à-point. Elle peut tolérer une ou plusieurs fautes temporelles de processeurs et de liens de communication. La tolérance aux fautes est obtenue hors-ligne et en deux phases. La première phase s'appuie

sur un formalisme de graphes pour transformer une spécification d'un algorithme sans redondance en une spécification avec redondances et relations d'exclusion. La deuxième phase, réalisée par une heuristique de distribution/ordonnancement statique, consiste à allouer spatialement et temporellement les composants logiciels de ce nouvel algorithme avec redondances sur les composants matériels de l'architecture, tout en garantissant que les contraintes de distribution, de temps réel et d'exclusion soient respectées.

2. La deuxième méthodologie consiste à générer automatiquement des distributions/ordonnancements tolérantes aux fautes, adaptées aux architectures matérielles distribuées et hétérogènes munies d'un réseau de communication composé uniquement de liaisons bus. Elle est basée sur la redondance hybride des composants logiciels et la fragmentation des données de communication en plusieurs paquets de données, pour tolérer une ou plusieurs fautes temporelles des processeurs et des bus de communication. Elle utilise une heuristique de distribution/ordonnancement statique qui consiste à allouer spatialement et temporellement les composants logiciels, redondants et fragmentés, de l'algorithme sur les composants matériels de l'architecture, tout en garantissant que les contraintes de distribution et de temps réel soient respectées.

3. La troisième méthodologie consiste à générer automatiquement des distributions et des ordonnancements fiables. La distribution/ordonnancement des composants logiciels de l'algorithme sur les composants matériels de l'architecture doit optimiser deux objectifs antagonistes, qui sont la minimisation de la durée d'exécution de l'algorithme sur l'architecture et la maximisation de la fiabilité de cette distribution/ordonnancement. Étant donné que, dans le cas général, la complexité du calcul de la fiabilité d'une distribution/ordonnancement est exponentielle, nous utilisons une méthode de transformation de graphe dans le but d'approximer ce calcul mais d'avoir une complexité polynomiale.

1.3 Structure du mémoire

Le mémoire est composé de trois parties :

- La première partie est constituée d'une présentation générale sur les systèmes réactifs et sur les moyens de sûreté de fonctionnement de ces systèmes.

  Dans le deuxième chapitre, nous donnons tout d'abord les concepts de bases et les terminologies liés aux systèmes réactifs. Ensuite, nous abordons le problème de la conception de ces systèmes, où nous ne présentons que les principes des méthodologies de conception basées sur la théorie d'ordonnancement, qui est ce qui nous intéresse le plus dans la suite du mémoire.

  Le troisième chapitre présente une technique qui permet de concevoir des systèmes sûrs de fonctionnement, qui est la tolérance aux fautes. Il présente aussi une mesure de probabilité, appelée fiabilité, qui est employé au sein de plusieurs méthodes pour de concevoir des systèmes fiables, c'est-à-dire des systèmes garantissant un niveau minimum de probabilité de bon fonctionnement. Nous présentons tout d'abord les terminologies liées à la tolérance aux fautes, ainsi que le problème de la génération de distributions/ordonnancements temps réel et tolérants aux fautes. Ensuite, nous abordons le problème de la définition d'un modèle de fiabilité pour le calcul de la fiabilité d'un tel système. À la fin de ce chapitre, nous présentons le problème de la génération de distributions/ordonnancements temps réel et fiables.

- La deuxième partie du mémoire est constituée d'une étude bibliographique, d'une présentation de nos modèles, et de nos trois méthodologies.

  Le quatrième chapitre présente les modèles pris en compte dans ce travail pour la modélisation des systèmes distribués, réactifs et embarqués. Ce sont le modèle d'algorithme, le modèle d'architecture et le modèle d'exécution. Le modèle d'algorithme décrit la spécification des composants logiciels de l'algorithme ; le modèle d'architecture décrit la spécification des composants physiques de l'architecture distribuée ; et le modèle d'exécution décrit le mode d'exécution des composants logiciels de l'algorithme sur les composants physiques de l'architecture. Puisque l'architecture est hétérogène, les caractéristiques d'exécution des composants logiciels sont potentiellement différents d'un composant physique à l'autre.

  Le cinquième chapitre présente un état de l'art sur les algorithmes de distribution/ordonnancement temps réel, tolérants aux fautes et fiables.

  Le sixième (resp. septième) chapitre présente la méthodologie que nous proposons pour résoudre le problème de la génération automatique de distributions/ordonnancements tolérantes aux fautes, adaptée aux architectures matérielles munies d'un réseau de communication composé uniquement de liaisons point-à-point (resp. bus).

  Le huitième chapitre présente notre méthodologie de génération de distributions/ordonnancements temps réel, qui consiste à optimiser deux critères antagonistes, qui sont la minimisation de la durée d'exécution de l'algorithme sur l'architecture et la maximisation de la fiabilité de la distribution/ordonnancement.

  Le neuvième chapitre présente une comparaison entre les trois méthodologies que nous proposons.

- La troisième partie présente les développements logiciels effectués pendant cette thèse. Le dixième chapitre présente le logiciel SynDEx implantant les trois méthodologies que nous proposons. Afin de mener une étude comparative de nos méthodologies, le onzième chapitre présente un générateur de graphes aléatoires.

Enfin, une conclusion résume nos contributions et présente quelques pistes de recherche futures.

Première partie

Concepts de base et terminologies

« Tout a un début, une existence, une fin »
Philippe STARCK

Chapitre 2

Introduction à la distribution et à l'ordonnancement temps réel

Résumé

Ce chapitre donne une présentation générale du problème de la distribution/ordonnancement pour des systèmes distribués réactifs embarqués. Après une présentation des terminologies liées à ce problème, ce chapitre donne une classification des algorithmes de distribution/ordonnancement temps réel qui peuvent résoudre ce problème. Enfin, il présente en détail un des algorithmes existant dans la littérature, et sur lequel est basé notre travail.

2.1 Introduction

Les systèmes réactifs sont de plus en plus présents dans de nombreux secteurs d'activité comme l'automobile, l'aéronautique, l'énergie, les télécommunications, le contrôle de processus industriel et le secteur militaire. Ces systèmes réalisent des tâches complexes qui sont souvent critiques, et ils sont soumis à des contraintes temporelles de l'environnement qu'ils doivent satisfaire. La conception de ces systèmes est souvent réalisée en deux étapes complémentaires : une étape de spécification et une étape d'implantation. La spécification consiste à définir l'algorithme à mettre en œuvre, l'architecture matérielle implantant de façon optimisée cet algorithme, et l'ensemble des contraintes temps réel, matérielles et de sûreté de fonctionnement. L'implantation optimisée consiste à trouver une mise en œuvre valide de l'algorithme sur l'architecture matérielle, c'est-à-dire qui satisfasse l'ensemble des contraintes. Il existe plusieurs façons pour réaliser une telle implantation, telles que le codage à la main de la spécification et la génération automatique de code.

Nous nous intéressons dans ce travail aux méthodes d'implantation basées sur la théorie d'ordonnancement, et plus particulièrement aux algorithmes de distribution et d'ordonnancement temps réel.

Le but principal de ce chapitre est de donner une idée générale sur les algorithmes de distributions et d'ordonnancements temps réel pour des systèmes réactifs. Nous commençons par quelques définitions de base sur ces systèmes. Ensuite, avant de présenter le problème de distribution et d'ordonnancement temps réel, nous présentons tout d'abord les terminologies liées à ce problème. Après, nous donnons une classification des algorithmes de distribution et d'ordonnancements qui peuvent résoudre ce problème. Enfin, nous présentons un des algorithmes de distribution et d'ordonnancement, existant dans la littérature, et qui nous intéresse plus particulièrement dans ce travail.

2.2 Définitions

2.2.1 Système réactif

Le concept de système réactif a été défini, dans la littérature, par plusieurs travaux [30, 47, 58]. La définition suivante est donnée par [30] :

Définition 1 (Système réactif) *Un système réactif est un système qui réagit continuement avec son environnement à un rythme imposé par cet environnement. Il reçoit, par l'intermédiaire de capteurs, des entrées provenant de l'environnement, appelées stimuli, réagit à tous ces stimuli en effectuant un certain nombre d'opérations et produit, grâce à des actionneurs, des sorties utilisables par l'environnement, appelées réactions ou commandes (figure 2.1).*

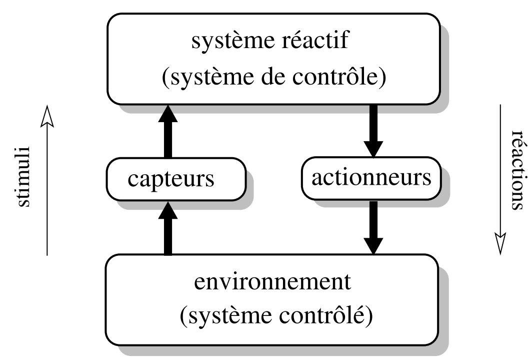


FIG. 2.1 – Modèle d'un système réactif.

Dans un système réactif, la validité d'une commande ne dépend pas uniquement de la validité de la valeur de son résultat, mais aussi de la validité de son instant de délivrance. Parfois dans la littérature, le système réactif est appelé *système de contrôle*, et l'environnement est appelé *système contrôlé* [47].

Un exemple très simple d'un système réactif est celui de la régulation de niveau d'eau dans un réservoir (figure 2.2). Dans cet exemple, l'environnement est constitué d'un réservoir d'eau, d'une vanne et de deux capteurs sensibles à la présence d'eau. Supposons qu'à l'instant t=0 le niveau d'eau dans le réservoir soit au niveau du capteur 1 et que la vanne soit ouverte. Le rôle de ce système est de maintenir le niveau d'eau entre les deux capteurs 1 et 2 : si le capteur 2 est mouillé le système doit envoyer une commande de fermeture de la vanne avant que le réservoir déborde, et si le capteur 1 est sec le système doit envoyer une commande d'ouverture de la vanne.

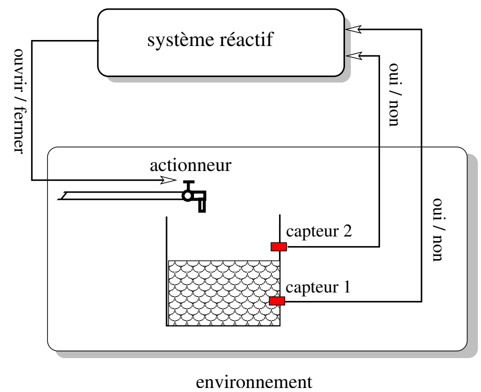


FIG. 2.2 – Exemple d'un système réactif.

Les exigences fonctionnelles et temporelles sont donc deux caractéristiques essentielles des systèmes réactifs qu'ils doivent respecter. Les exigences fonctionnelles imposent au système de produire des résultats corrects du point de vue de leurs valeurs. Les exigences temporelles imposent au système de produire ces résultats à temps, c'est-à-dire que le système doit réagir à chaque événement de l'environnement externe avant l'échéance imposée par l'environnement pour le prochain événement.

Suivant les exigences temporelles d'un système réactif, on peut distinguer deux classes de ces systèmes :

- **Systèmes réactifs à contraintes temps réel strictes** : Ces systèmes doivent impérativement respecter les contraintes de temps imposées par l'environnement, et le non respect d'une contrainte temporelle peut avoir des conséquences catastrophiques. Le système de contrôle de trafic aérien et de conduite de missile sont deux exemples de ces systèmes.
- **Systèmes réactifs à contraintes temps réel souples** : Ces systèmes se caractérisent par leurs souplesses envers les contraintes temps réel imposées par l'environnement, il s'agit d'exécuter dans les meilleurs délais les fonctions. Le non respect d'une contrainte temporelle peut être acceptable par ces systèmes.

2.2.2 Système distribué et système embarqué

Définition 2 (Système distribué) *Un système distribué est tel que son architecture matérielle est distribuée, c'est-à-dire composée de plusieurs machines multiprocesseurs ou monoprocesseur re-*

liées entre elles par un ensemble de moyens de communication (mémoire partagée, bus, liaison point-à-point ...).

De plus, dans le cas général, cette architecture distribuée est *hétérogène*, c'est-à-dire que les processeurs (resp. média de communication) ont des caractéristiques physiques différentes.

Définition 3 (Système embarqué) *Un système embarqué se caractérise par ses ressources limitées, telles que la mémoire, la consommation électrique, la taille, le poids, la puissance de calcul ...*

Le terme important dans cette définition est *limitées*. Par exemple, la taille du système peut être grande (par exemple, un avion) aussi bien que petite (par exemple, un PDA), mais dans les deux cas elle est limitée.

2.3 Spécification des systèmes distribués réactifs embarqués

Nous nous intéressons dans ce travail aux systèmes distribués réactif embarqués à contraintes temps réel strictes. La spécification de ces systèmes est réalisée en trois phases complémentaires et dépendantes, qui sont la spécification fonctionnelle, la spécification architecturale et la spécification des contraintes. La spécification fonctionnelle consiste à définir l'algorithme avec ses exigences fonctionnelles ; la spécification architecturale consiste à définir l'architecture matérielle qui doit implanter l'algorithme ; et la spécification des contraintes consiste à attribuer des propriétés temporelles et matérielles à l'exécution de l'algorithme sur l'architecture.

Remarque 1 *Pour des raisons de lisibilité, dans la suite de ce mémoire, nous écrivons « système réactif » au lieu de « système distribué réactif embarqué à contraintes temps réel strictes ».*

2.3.1 L'algorithme

L'algorithme d'un système réactif représente les fonctions nécessaires au contrôle et à la commande de l'environnement. Il est composé d'un ensemble d'entités informatiques, que nous appelons *composants logiciels* ; chacun assure une fonctionnalité spécifique du système, telle que la commande d'une vanne de régulation. Un composant logiciel peut être composé de plusieurs sous-composants logiciels qui communiquent pour accomplir une fonctionnalité du système.

L'algorithme peut être modélisé par un graphe flot de données. Les noeuds représentent les composants logiciels, et les arcs représentent les dépendances de données entre ces composants. Ces arcs induisent un ordre partiel d'exécution sur les opérations, appelé parallélisme potentiel. Une dépendance de données entre deux composants o_1 et o_2, notée $(o_1 \triangleright o_2)$, signifie que o_2 commencera son exécution après avoir reçu les données de sortie de o_1. La figure 2.3 est un exemple d'un algorithme composé de six composants logiciels $(In_1, In_2, A, B, Out_1, Out_2)$ et de cinq dépendances de données $(In_1 \triangleright A, In_1 \triangleright B, In_2 \triangleright Out_2, A \triangleright Out_1, B \triangleright Out_1)$. Un composant logiciel sans prédécesseurs désigne une fonctionnalité de lecture des données d'un capteur (In_1 et In_2 dans la figure 2.3), tandis qu'un composant sans successeurs désigne une fonctionnalité de commande d'un actionneur (Out_1 et Out_2 dans la figure 2.3).

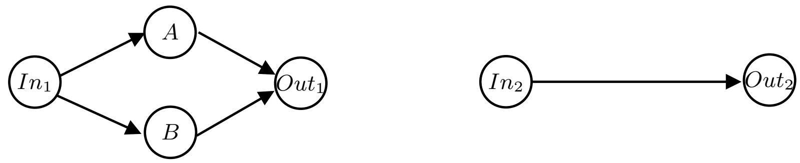


FIG. 2.3 – Exemple d'un algorithme.

2.3.2 L'architecture matérielle

Dans un système distribué, l'architecture matérielle est composée de plusieurs machines mono ou multiprocesseurs, de moyens de communication, et de plusieurs capteurs et actionneurs. Chaque machine dispose de sa propre horloge et de sa propre mémoire. La spécification de l'architecture matérielle consiste donc à caractériser tous les composants de l'architecture et à choisir la topologie du réseau de communication :

- **Processeurs** : la spécification de l'algorithme ne peut être complètement indépendante de l'architecture, puisque les attributs temporels des composants logiciels sont en relation directe avec le type des processeurs utilisés. Ainsi, la durée d'exécution d'un composant entité est différente d'un processeur à un autre puisque l'architecture est hétérogène.
- **Capteurs et actionneurs** : les systèmes réactifs utilisent des capteurs et des actionneurs pour interagir avec l'environnement extérieur qu'ils contrôlent. Étant donné que l'architecture matérielle est distribuée, le choix de l'emplacement physique de ces capteurs/actionneurs sur l'architecture est important dans la conception de ces systèmes.
- **Moyen de communication** : les processeurs de l'architecture distribuée communiquent entre eux en utilisant deux types de communications : communication par passage de message, et communication par partage de données [81]. Pour pouvoir gérer les communications inter-processeurs entre des composants logiciels dépendants, placées sur des processeurs distincts, il est donc nécessaire de connaître le type des communications et le type des média de communication (mémoire partagée, bus, lien point-à-point . . .).

La figure 2.4 est un exemple d'une architecture distribuée composée de cinq processeurs (P_1 à P_5) et de trois mémoires locales (M_1 à M_3). Les processeurs communiquent entre eux via un réseau de communication composé d'une mémoire partagée et d'une liaison physique de communication.

2.3.3 Les contraintes temporelles et matérielles

Les contraintes temporelles s'expriment, pour chaque composant logiciel de l'algorithme, en termes de sa date de début d'exécution, et de sa date de fin d'exécution avant une échéance. La date de début d'exécution est liée à l'occurrence de certains événements ou à la satisfaction de certaines conditions (par exemple, la réception de données), tandis que la date de fin d'exécution est liée au comportement exigé par l'environnement. Les contraintes temporelles attribuées aux composants de l'algorithme peuvent être des mesures exactes, moyennes, ou majoritaires ; ceci dépend des moyens utilisés pour obtenir ces mesures.

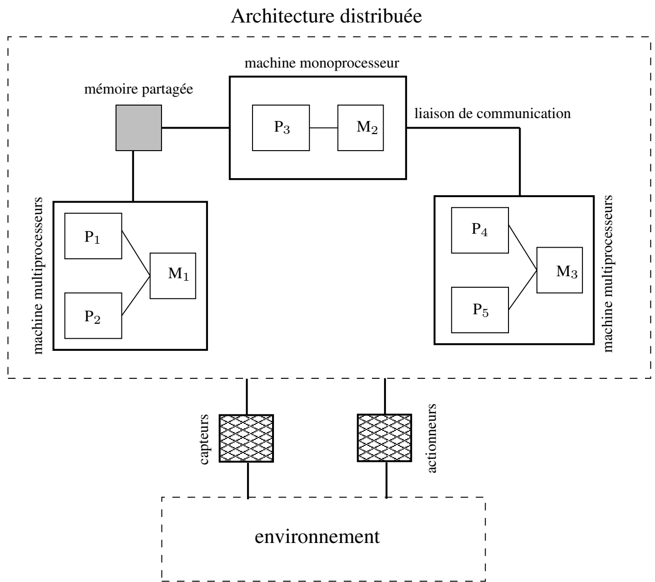


FIG. 2.4 – Exemple d'une architecture distribuée.

La spécification de l'algorithme n'est pas complètement indépendante de l'architecture : par exemple, afin de réduire le câblage, certains capteurs et actionneurs doivent être gérés par des processeurs spécifiques de l'architecture. Il est donc nécessaire de spécifier des contraintes matérielles pour chaque composant ou sous-composant de l'algorithme, c'est-à-dire, attribuer à chacun un ou plusieurs processeurs de l'architecture matérielle qui peuvent l'exécuter.

2.4 Problème de distribution et d'ordonnancement temps réel

Une fois que les modèles d'algorithme et d'architecture ont été spécifiés, et que les contraintes temporelles et matérielles ont été spécifiées, il est ensuite possible de formuler précisément le problème de la recherche d'une implantation de l'algorithme sur l'architecture qui satisfasse les contraintes temporelles et matérielles.

2.4.1 Terminologies

Nous définissons dans cette section quelques concepts qui seront utilisés tout au long de ce mémoire.

Définition 4 (Ordonnancement temps réel) *L'ordonnancement est le terme informatique désignant le mécanisme permettant de réaliser la sélection, parmi plusieurs composants de l'algorithme, de celui qui va obtenir l'utilisation d'un composant de l'architecture pour s'exécuter, de manière à optimiser un ou plusieurs critères. L'ordonnancement temps réel a une particularité qui nécessite de respecter des contraintes de temps réel. Le processus qui réalise une telle sélection, et* *donc qui définit un ordre d'exécution entre les composants de l'algorithme, est appelé algorithme d'ordonnancement.*

Définition 5 (Ordonnancement statique et dynamique) *Si l'ordre d'exécution des composants logiciels d'un algorithme ne change pas pendant l'exploitation du système, alors il s'agit d'un ordonnancement statique. Dans le cas contraire, il s'agit d'un ordonnancement dynamique.*

Définition 6 (Fonction de coût) *Le rôle d'une fonction de coût est d'associer un poids à chaque composant logiciel, et ceci afin de calculer un ordre d'exécution entres ces composants.*

Définition 7 (Algorithme de distribution et d'ordonnancement temps réel) *C'est un algorithme d'ordonnancement temps réel qui doit réaliser, en plus d'une opération de sélection d'un composant logiciel, une opération de sélection du composant matériel qui peut exécuter le composant logiciel sélectionné. Ce type d'algorithme est nécessaire dès que l'architecture est distribuée.*

2.4.2 Présentation du problème

Le problème de la recherche d'une distribution (allocation spatiale) et d'un ordonnancement (allocation temporelle) des composants logiciels de l'algorithme sur les composants matériels de l'architecture distribuée, avec contraintes temps réel, est un *problème de distribution et d'ordonnancement temps réel* [82]. La solution à ce problème consiste à chercher l'existence d'une allocation valide, c'est-à-dire qui satisfasse les contraintes temporelles et matérielles. Ce problème de recherche devient un problème d'optimisation lorsqu'il s'agit de rechercher une allocation valide qui doit en plus optimiser un certain critère (par exemple, minimiser les coûts de communication) ; si celle-ci existe, elle est optimale. Le processus qui réalise cette tâche de recherche et d'optimisation est appelé *algorithme de distribution et d'ordonnancement temps réel*. Ce problème d'optimisation peut être soit NP-difficile soit polynômial [28]. Dans ce dernier cas, si une solution optimale à ce problème existe, l'algorithme est capable de la trouver dans un temps polynômial par rapport à la taille du problème (usuellement le nombre de composants logiciels de l'algorithme), tandis que dans l'autre cas, l'algorithme la trouve mais dans un temps exponentiel.

Remarque 2 *Pour des raisons de lisibilité, dans la suite de ce mémoire, nous désignons par le terme « distribution/ordonnancement » une distribution et un ordonnancement temps réel.*

Enfin, le problème de distribution/ordonnancement des composants logiciels d'un algorithme sur les composants matériels d'une architecture est formalisé comme suit :

Problème 1 *Étant donnés :*

- *une architecture matérielle hétérogène $\mathcal{A}rc$ composée de n composants matériels :*

$$\mathcal{A}rc = \{p_1, ..., p_n\}$$

- *un algorithme $\mathcal{A}lg$ composé de m composants logiciels :*

$$\mathcal{A}lg = \{o_1, ..., o_m\}$$

- *des coûts d'exécution $\mathcal{E}xe$ des composants de $\mathcal{A}lg$ sur les composants de $\mathcal{A}rc$,*
- *des contraintes temps réel $\mathcal{R}tc$ et matérielles $\mathcal{D}is$,*
- *un ensemble de critères à optimiser,*

il s'agit de trouver une application $\mathcal{A}$ qui associe chaque composant o_i de $\mathcal{A}lg$ à un composant p_j de $\mathcal{A}rc$, et qui lui assigne un ordre d'exécution t_k sur son composant matériel :

$$\begin{array}{rccl} \mathcal{A}: & \mathcal{A}lg & \longrightarrow & \mathcal{A}rc \\ & o_i & \longmapsto & \mathcal{A}(o_i) = (p_j, t_k) \end{array}$$

qui doit satisfaire $\mathcal{R}tc$ et $\mathcal{D}is$, et optimiser les critères spécifiés.

2.5 Algorithmes de distribution et d'ordonnancement temps réel

La tâche principale d'un algorithme de distribution/ordonnancement est de choisir, à un instant donné durant l'exécution du système, un composant matériel pour le composant logiciel le plus prioritaire. Suivant ce choix, on peut donc distinguer dans la littérature deux grandes classes d'algorithmes qui peuvent résoudre le problème 1. Nous les présentons ici.

2.5.1 Algorithmes hors-ligne et en-ligne

Les algorithmes hors-ligne supposent que les caractéristiques temporelles de tous les composants logiciels de l'algorithme sont connues et fixées à l'avance, c'est-à-dire, avant la mise en exploitation du système. Ceci permet à ces algorithmes de réaliser une allocation spatiale et temporelle des composants logiciels sur les composants matériels avant l'exécution du système ; cette allocation est statique puisque l'ordre d'exécution des composants logiciels ne change pas au cours de l'exécution du système. Ces algorithmes permettent de concevoir des *systèmes prédictifs*, c'est-à-dire que les contraintes temporelles peuvent être vérifiées et validées avant la mise en exploitation du système.

Dans certains systèmes réactifs, certaines caractéristiques temporelles des composants logiciels ne sont connues et fixées qu'au moment de l'exécution du système. Par exemple, la date de début d'exécution d'un composant logiciel assurant une fonctionnalité d'alarme, en cas d'incident, n'est connue qu'au moment de la présence d'un incident durant l'exécution du système. Donc, l'allocation spatiale et temporelle d'un tel composant logiciel sur un composant matériel doit être effectuée durant l'exécution du système, et ceci uniquement à l'instant où toutes ses caractéristiques temporelles sont disponibles. Les algorithmes réalisant ce type d'allocation sont appelés algorithmes en-ligne.

2.5.2 Algorithmes exacts et approchés

Les algorithmes hors-ligne et en-ligne qui trouvent une solution optimale, si celle-ci existe, au problème de la distribution/ordonnancement temps réel, reposent dans le pire des cas sur l'exploration de tout l'espace de recherche. Ces algorithmes de distribution/ordonnancement appartiennent à la classe des *algorithmes exacts* puisqu'ils renvoient toujours une solution optimale si celle-ci existe. Cependant, dans le cas général, ce problème est NP-difficile et la complexité est exponentielle.

La recherche de l'optimalité de la solution pour ce problème de distribution/ordonnancement n'est pas usuellement une contrainte d'un système réactif. C'est pourquoi, pour résoudre ce problème NP-difficile dans un temps polynômial, plusieurs algorithmes heuristiques ont été développés dans la littérature pour chercher une solution valide et si possible proche de la solution optimale. Ces algorithmes de distribution et d'ordonnancement appartiennent à la classe des *algorithmes approchés*.

2.6 Algorithme de distribution et d'ordonnancement de SYNDEX

Parmi les algorithmes de distribution/ordonnancement temps réel proposés dans la littérature, nous ne nous intéressons dans ce travail qu'aux algorithmes hors-ligne approchés. L'heuristique de distribution/ordonnancement implanté dans SYNDEX[1] est un algorithme de ce type proposé dans sa première version en 1991 [51]. Elle a depuis été beaucoup améliorées afin de pouvoir traiter des problèmes réaliste de types industriels où l'on cherche à distribuer des algorithmes de plusieurs centaines de composants logiciels sur quelques dizaines de processeurs et média de communication.

2.6.1 L'algorithme, l'architecture et les contraintes temporelles et matérielles

L'algorithme est spécifié par un graphe flot de donnée, appelé graphe d'algorithme $\mathcal{A}lg$. Les sommets du graphe d'algorithme sont les opérations de l'algorithme et les arcs sont les dépendances de données entre opérations. Une opération ne peut s'exécuter que lorsque ses données d'entrée sont présentes. Elle consomme ses données d'entrée et produit des données en sortie, qui sont ensuite utilisées par ses successeurs. Une opération sans prédécesseur (resp. sans successeur)

[1] SYNDEX est un environnement graphique interactif de développement pour applications temps réel. `http://www.syndex.org`

représente une interface d'entrée, c'est-à-dire un capteur, (resp. une interface de sortie, c'est-à-dire un actionneur) avec l'environnement.

L'architecture matérielle est spécifiée par un graphe non orienté appelé graphe d'architecture $\mathcal{A}rc$. Les sommets désignent les processeurs et les mémoires, les arêtes désignent les liaisons physiques entre mémoires et processeurs. Chaque processeur est composé d'une unité de calcul, d'une mémoire locale, et d'une ou plusieurs unités de communication pour communiquer avec les périphériques ou d'autres processeurs.

À chaque opération o_i du graphe d'algorithme $\mathcal{A}lg$ est associé un coût d'exécution $\mathcal{E}xe(o_i,p_k)$ sur chaque processeur p_k du graphe d'architecture $\mathcal{A}rc$. Ce temps d'exécution désigne une borne supérieure (WCET) du temps d'exécution de l'opération sur chaque processeur. Il est déterminé par le concepteur du système en utilisant des méthodes statiques ou dynamiques [18]. L'architecture étant hétérogène, ces coûts d'exécutions peuvent être distincts pour un même composant logiciel de $\mathcal{A}lg$. De plus, à chaque dépendance de données $(o_i \rhd o_j)$ de $\mathcal{A}lg$ est associé un coût de transfert de données $\mathcal{E}xe(o_i \rhd o_j,l_k)$ sur le lien de communication l_k de $\mathcal{A}rc$.

Les contraintes matérielles sont spécifiées par l'association de la valeur ∞ à $\mathcal{E}xe(o_i,p_k)$, ce qui signifie que l'opération o_i ne peut pas être implantée sur le processeur p_k. Enfin, une seule contrainte temps réel $\mathcal{R}tc$ est prise en compte dans l'algorithme de distribution/ordonnancement, qui est la contrainte de la latence, c'est-à-dire que la longueur de la distribution/ordonnancement du graphe d'algorithme sur le graphe d'architecture doit être inférieure à un seuil défini par la latence.

Remarque 3 *Les deux graphes d'algorithme et d'architecture, et les deux contraintes matérielles et temporelles seront présentés en détail dans le chapitre 4.*

2.6.2 Présentation de l'algorithme de distribution et d'ordonnancement

Le but principal de cette heuristique de distribution/ordonnancement est de chercher une allocation spatiale et temporelle du graphe d'algorithme $\mathcal{A}lg$ sur le graphe d'architecture$\mathcal{A}rc$, tout en respectant la contrainte temps réel $\mathcal{R}tc$.

Avant de présenter l'heuristique, nous donnons tout d'abord les notations suivantes qui seront utilisées par cette heuristique et aussi dans le reste de ce travail :

- $pred(o_i)$: l'ensemble des opérations prédécesseurs de l'opération o_i,
- $succ(o_i)$: l'ensemble des opérations successeurs de l'opération o_i,
- $\mathcal{E}xe(o_i, p_j)$: le coût d'exécution de o_i sur l'opérateur p_j,
- $X^{(n)}$: l'exposant n entre parenthèses désigne l'étape de l'heuristique, c'est-à-dire après avoir alloué la n^{ieme} opération ; donc, $X^{(n)}$ désigne l'ensemble X à l'étape n de l'heuristique,
- $O_{cand}^{(n)}$: la liste des opérations *candidates* ; une opération de $\mathcal{A}lg$ est dite candidate si elle est implantable, c'est-à-dire que tous ses prédécesseurs sont déjà alloués,
- $O_{fin}^{(n)}$: la liste des opérations déjà allouées,

- $St^{(n)}_{o_i,p_j}$: la date de début au-plus-tôt de o_i sur p_j, depuis le début [81],
- $st^{(n)}_{o_i,p_j}$: la date de début au-plus-tard de o_i sur p_j, depuis le début [81],
- $\overline{st}^{(n)}_{o_i}$: la date de début au-plus-tard de o_i, depuis la fin [81] ;

L'heuristique de distribution/ordonnancement est un algorithme glouton de type ordonnancement de liste [83], basée sur une fonction de coût appelée la *pression d'ordonnancement*, dont l'objectif est de minimiser la longueur de la distribution/ordonnancement.

Définition 8 (Longueur d'une distribution/ordonnancement) *La longueur de la distribution/ordonnancement d'un graphe d'algorithme sur un graphe d'architecture, noté $R^{(n)}$, est la durée d'exécution de ce graphe d'algorithme sur ce graphe d'architecture, c'est-à-dire la date de terminaison du dernier processeur en exécution.*

Définition 9 (Pression d'ordonnancement) *La pression d'ordonnancement, notée $\sigma^{(n)}_{o_i,p_j}$, est une fonction de coût qui induit des priorités d'ordonnancement entre les opérations de $\mathcal{A}lg$. Elle mesure à la fois la marge d'ordonnancement $F^{(n)}$ et l'allongement $P^{(n)}$ de la longueur de la distribution/ordonnancement. Elle est calculée pour chaque opération candidate o_i sur chaque processeur p_j par :*

$$\sigma^{(n)}_{o_i,p_j} = P^{(n)}_{o_i,p_j} - F^{(n)}_{o_i,p_j} \tag{2.1}$$

Définition 10 (Pénalité d'ordonnancement) *La pénalité d'ordonnancement, notée $P^{(n)}_{o_i,p_j}$, est une fonction qui mesure l'allongement de la longueur de la distribution/ordonnancement $R^{(n)}_{o_i,p_j}$ après avoir alloué o_i sur p_j à la n^{ieme} étape de l'heuristique, en tenant compte des coûts de communication engendrées par l'allocation. Elle est définie par :*

$$P^{(n)}_{o_i,p_j} = R^{(n)}_{o_i,p_j} - R^{(n-1)} \tag{2.2}$$

Définition 11 (Flexibilité d'ordonnancement) *La flexibilité d'ordonnancement, notée $F^{(n)}_{o_i,p_j}$, est une fonction qui mesure la marge d'ordonnancement de o_i sur p_j à la n^{ieme} étape de l'heuristique. Elle est définie par :*

$$F^{(n)}_{o_i,p_j} = st^{(n)}_{o_i,p_j} - St^{(n)}_{o_i,p_j} \tag{2.3}$$

D'après l'équation suivante [81] :

$$st^{(n)}_{o_i,p_j} = R^{(n)}_{o_i,p_j} - \overline{st}^{(n)}_{o_i} \tag{2.4}$$

et les équations (2.1), (2.2) et (2.3), la pression d'ordonnancement est définie par :

$$\sigma^{(n)}_{o_i,p_j} = St^{(n)}_{o_i,p_j} + \overline{st}^{(n)}(o_i) - R^{(n-1)} \tag{2.5}$$

L'heuristique de distribution/ordonnancement a la forme suivante :

ALGORITHME

- ***Entrées =*** *$\mathcal{A}lg$, $\mathcal{A}rc$, $\mathcal{E}xe$, $\mathcal{D}is$ et $\mathcal{R}tc$;*
- ***Sortie =*** *Distribution/ordonnancement statique de $\mathcal{A}lg$ sur $\mathcal{A}rc$ en fonction de $\mathcal{E}xe$ et $\mathcal{D}is$ et qui satisfait $\mathcal{R}tc$, ou un message d'échec ;*

INITIALISATION

Initialiser la liste des opérations candidates, et la liste des opérations déjà allouées :

$O^{(1)}_{cand}$:= {*opérations de* $\mathcal{A}lg$ *sans prédécesseurs*} ;
$O^{(1)}_{fin} := \emptyset$;

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

Tant que $O^{(n)}_{cand} \neq \emptyset$ ***faire***

SÉLECTION

- *Calculer pour chaque candidate $o_{cand} \in O^{(n)}_{cand}$ et chaque processeur $p_j \in \mathcal{A}rc$ la pression d'ordonnancement (équation (2.5)) ;*
- *Sélectionner pour chaque candidate o_{cand} le processeur p_{best} qui minimise la pression d'ordonnancement ;*
- *Sélectionner le meilleur couple (o_{best}, p_{best}) qui maximise la pression d'ordonnancement ;*

DISTRIBUTION ET ORDONNANCEMENT

- *Placer cette candidate o_{best} sur le processeur p_{best} (allocation spatiale) ;*
- *Placer et ordonnancer toutes les communications nécessaires à ce placement : $(o_i \triangleright o_{best}) \forall o_i \in pred(o_{best})$;*
- *Ordonnancer o_{best} sur le processeur p_{best} (allocation temporelle) ;*

VÉRIFICATION DES CONTRAINTES TEMPORELLES

- **si** $\left(R^{(n)}_{o_i,p_j} > \mathcal{R}tc\right)$ **alors** *terminer et répondre « échec » ;*

MISE À JOUR

- *Mettre à jour la liste des opérations candidates et déjà placées :*

  $O^{(n+1)}_{fin} := O^{(n)}_{fin} \cup \{o_{best}\}$;
  $O^{(n+1)}_{cand} := O^{(n)}_{cand} - \{o_{best}\} \cup \left\{ o \in succ(o_{best}) \mid pred(o) \subseteq O^{(n+1)}_{fin} \right\}$;

Fin tant que

FIN DE L'ALGORITHME

Les grandes lignes de l'heuristique sont les suivantes :

- à chaque étape, une liste d'opérations candidates $O^{(n)}_{cand}$ est établie.
- Pour chaque opération o_i de $O^{(n)}_{cand}$, une pression d'ordonnancement $\sigma^{(n)}_{o_i,p_j}$ sur chaque proces-

seur p_j de $\mathcal{A}rc$ est calculée. La fonction de coût de la pression d'ordonnancement vise à minimiser la longueur de la distribution/ordonnancement.

- La liste d'opérations candidates $O_{cand}^{(n)}$ est triée par pression d'ordonnancement décroissante ($\max \sigma_{o_i,p_j}^{(n)}$),
- puis, la première opération suivant cet ordre est placée et ordonnancée sur le processeur déterminé par la fonction de coût,
- ensuite, ce processus d'allocation est répété pour toutes les opérations restantes, jusqu'à ce qu'il n'en reste plus.

2.7 Conclusion

Nous avons présenté dans ce chapitre le problème de distribution/ordonnancement temps réel pour des systèmes distribués réactifs embarqués. Nous avons vu que ce problème peut être résolu par des algorithmes de distribution/ordonnancement temps réel, et qu'il en existe deux grandes classifications de ces algorithmes : d'une part les algorithmes hors-ligne ou en-ligne, et d'autre part les algorithmes exacts ou approchés. Un exemple d'un algorithme de distributions/ordonnancement hors-ligne/approché a été présenté. Dans la suite de ce travail, nous ne parlerons que des algorithmes de distribution/ordonnancement hors-ligne et approchés.

« Deux sûretés valent mieux qu'une. »
JEAN DE LA FONTAINE

Chapitre 3

Tolérance aux fautes et fiabilité des systèmes réactifs

Résumé

Ce chapitre présente une technique qui permet de concevoir des systèmes sûrs de fonctionnement, qui est la tolérance aux fautes. Elle permet à un système de continuer à délivrer un service conforme à sa spécification en présence de fautes. Il présente aussi une mesure de probabilité, appelée fiabilité, qui est employé au sein de plusieurs méthodes afin de concevoir des systèmes fiables, c'est-à-dire des systèmes garantissant un niveau minimum de probabilité de bon fonctionnement.

3.1 Introduction

La théorie de l'ordonnancement est l'un des moyens qui permet d'obtenir des systèmes réactifs. Elle s'intéresse à l'allocation optimale d'un algorithme sur une architecture qui doit satisfaire des contraintes temporelles. Cependant, le respect des contraintes temporelles est une condition nécessaire mais pas suffisante pour le bon fonctionnement d'un tel système, puisque la présence de certaines fautes est inévitable, telles que les fautes de conception des composants logiciels, appelées « bugs », et les fautes de conception des composants matériels, appelées « errata » [8]. Au vu des conséquences catastrophiques (perte d'argent, de temps, ou pire de vies humaines) que pourrait entraîner une faute dans un système réactif critique, les techniques de sûreté de fonctionnement [50, 9] sont vitales dans la conception de ces systèmes. Elles permettent la conception de systèmes sûrs de fonctionnement.

La *tolérance aux fautes* est l'un des moyens utilisés dans la littérature par plusieurs méthodes pour concevoir des systèmes sûrs de fonctionnement, que nous appelons dans la suite « systèmes tolérants aux fautes ». Elle permet à un système de continuer à délivrer un service conforme à sa spécification en présence de fautes. D'autres méthodes utilisent une mesure de probabilité, appelée fiabilité, pour concevoir aussi des systèmes sûrs de fonctionnement, que nous appelons dans la suite « systèmes fiables ». La fiabilité permet d'évaluer stochastiquement le bon fonctionnement d'un système.

Dans ce chapitre, nous présentons tout d'abord les terminologies liées à la tolérance aux fautes, et ses deux phases de réalisation. Ensuite, nous abordons le problème de la définition d'un modèle de fiabilité pour le calcul de la fiabilité d'un tel système. Enfin, nous présentons la relation existant entre la théorie de l'ordonnancement et le calcul de la fiabilité.

3.2 Tolérance aux fautes des systèmes réactifs

3.2.1 Terminologies

Définition 12 (Faute) *La faute dans un système informatique représente soit un défaut d'un composant physique, ou soit un défaut d'un composant logiciel de ce système. Elle peut être créée de manière intentionnelle ou accidentelle, à cause des phénomènes physiques ou à cause des imperfections humaines. Durant l'exécution du système, la faute reste dormante jusqu'à ce qu'un événement intentionnel ou accidentel provoque son activation.*

Définition 13 (Erreur) *L'activation d'une faute durant l'exploitation du système peut se manifester par la présence d'un état interne erroné dans ce système. Sous des circonstances particulières cet état interne erroné peut conduire à une* ***erreur****, c'est-à-dire, à un résultat incorrect ou imprécis.*

Définition 14 (Défaillance) *Une erreur peut changer le comportement d'un système et provoquer le non respect de sa spécification : c'est une* ***défaillance*** *du système.*

Donc, la défaillance d'un système est la conséquence d'une erreur, et l'erreur est la conséquence d'une faute activée. En plus, étant donné qu'un système informatique est souvent composé de plusieurs sous-systèmes[1], la défaillance d'un sous-système peut créer et/ou activer une faute dans un autre sous-système, ou dans le système lui même. La relation entre ces trois termes faute, erreur et défaillance relativement aux sous systèmes du système peut être représentée par la figure 3.1.

Suivant la persistance temporelle d'une faute, on distingue trois types de fautes qui peuvent affecter un système : fautes permanentes, transitoires et intermittentes.

Définition 15 (Faute permanente) *Une faute permanente se caractérise par sa durée permanente, une fois activée, durant l'exploitation du système. Elle persiste donc indéfiniment (jusqu à réparation) après son occurrence.*

[1]Par exemple, un processeur et un composant logiciel sont des sous-systèmes.

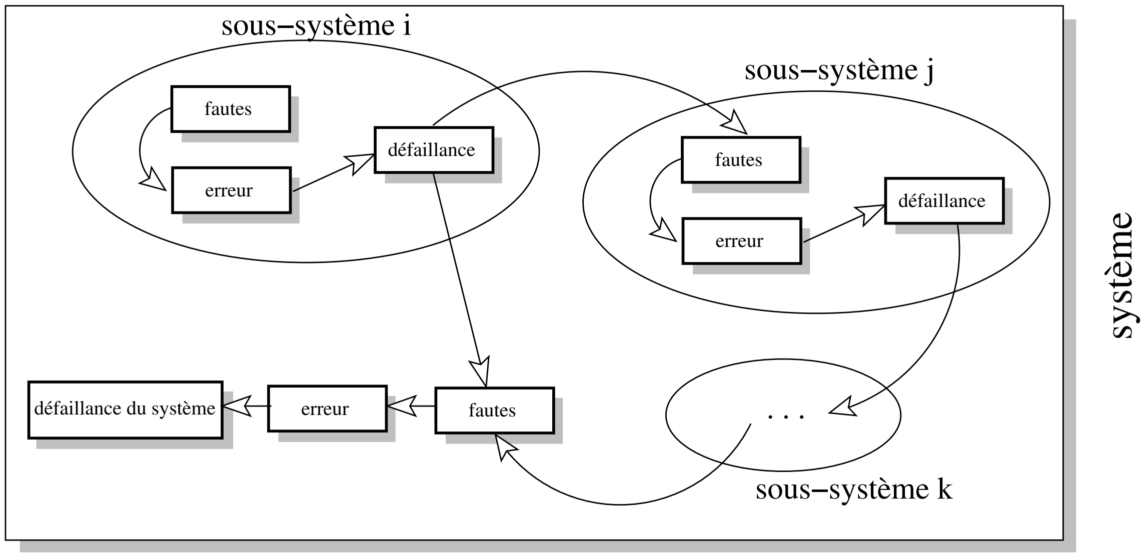


FIG. 3.1 – Relation entre faute, erreur et défaillance

Une faute de conception d'un composant matériel est un exemple typique de faute permanente. Le processus de conception et de fabrication des composants matériels modernes peut avoir des conséquences sur l'occurrence des fautes permanentes. Par exemple, l'étude réalisée sur la conception des mémoires DRAM (Dynamic Random Access Memory) montre que l'augmentation de la capacité d'une mémoire RAM à 32 fois sa capacité initiale provoque une augmentation de 50% du taux d'occurrence de fautes permanentes [19].

Définition 16 (Faute transitoire) *Une faute transitoire se caractérise par sa durée limitée, une fois activée, durant l'exploitation du système.*

Les fautes transitoires sont souvent observées dans les systèmes de communication, où la présence des radiations électromagnétiques peut corrompre les données envoyées sur une liaison physique de communication. Ceci provoque une faute transitoire qui ne dure que la période de la présence de ces radiations.

Définition 17 (Faute intermittente) *Une faute intermittente est une faute transitoire qui se reproduit sporadiquement.*

Suivant les exigences fonctionnelles et temporelles d'un système réactif, la défaillance de ce système peut être la conséquence de deux sources de fautes : fautes fonctionnelles (ou fautes de valeur) et fautes temporelles.

Définition 18 (Faute fonctionnelle) *La valeur délivrée par le système est fausse, c'est-à-dire qu'elle n'est pas conforme à sa spécification, ou elle est en dehors de l'intervalle des valeurs attendues.*

Définition 19 (Faute temporelle) *L'instant auquel la valeur est délivrée est en dehors de l'intervalle de temps spécifié. Dans ce cas, la valeur est considérée soit temporellement délivrée* trop tôt*, soit* trop tard*, soit* infiniment tard *(*jamais délivrée*). La faute temporelle dans le cas où la valeur n'est jamais délivrée est appelée «* ***faute par omission*** *».*

Enfin, le concept de la tolérance aux fautes a été défini, dans la littérature, par plusieurs travaux [7, 13, 42, 50, 77, 79]. La définition suivante est la définition originale donnée par Algirdas Avizienis [7] en 1967 :

Définition 20 (Tolérance aux fautes) *On dit qu'un système informatique est tolérant aux fautes si ses programmes peuvent être exécutés correctement même en présence de fautes.*

3.2.2 Algorithmes de tolérance aux fautes

La tolérance aux fautes est réalisée en deux phases complémentaires : une phase de détection des erreurs et une phase de traitement des erreurs. Nous appelons dans la suite le processus qui regroupe ces deux phases « *algorithme de tolérance aux fautes* ».

3.2.2.1 Phases d'un algorithme de tolérance aux fautes

A. Détection des erreurs

La tâche la plus importante d'un algorithme de tolérance aux fautes est la détection des erreurs, puisque c'est d'elle que dépend la réussite de l'algorithme. La détection des erreurs permet d'identifier le type et l'origine des erreurs. Cette détection peut être faite soit au niveau de l'environnement du système, soit au niveau de l'application du système [13]. Au niveau de l'environnement, c'est l'exécutif de l'application qui se charge de détecter certaines erreurs, qui peuvent être par exemple de type « division par zéro », « erreur d'entrée/sortie », « accès interdit au périphérique ». Au niveau de l'application, ce sont les composants redondants[2] qui se chargent de réaliser cette tâche de détection. Les techniques de détection d'erreurs au niveau de l'application sont nombreuses. Parmi les techniques de base, on trouve la comparaison des résultats de composants répliqués et la vérification des temps de délivrance des résultats. Enfin, la réussite d'une telle technique de détection des erreurs dépend de deux paramètres qui sont la *latence* (délai entre la production et la détection de l'erreur), et le *taux de couverture* (pourcentage d'erreurs détectées).

B. Traitement des erreurs

Cette phase consiste à traiter les états erronés (ou erreurs) détectés par la première phase, à l'aide d'une des deux techniques de base suivantes :

[2]Nous désignons ici par composant redondant tout composant du système qui ne fait pas partie de sa spécification initiale (avant la tolérance aux fautes), et qui est donc « en plus ».

- *recouvrement des erreurs* : le recouvrement consiste à remplacer l'état erroné par un état correct. Il utilise soit la technique de la reprise, soit la technique de la poursuite. La reprise consiste à mettre le système dans un de ses états précédents corrects, et la poursuite consiste soit en la reconstitution d'un état correct, soit en la reconstitution partielle d'un état correct qui permet au système de fonctionner en mode dégradé.
- *compensation des erreurs* : la compensation consiste en la reconstruction totale d'un état correct en utilisant un ensemble d'informations redondantes existantes dans le système.

Le choix entre ces deux techniques est un compromis entre plusieurs facteurs, tels que la complexité du système, les contraintes temporelles et matérielles et la criticité du système. Étant donné que nous visons des systèmes réactifs embarqués critiques, nous ne nous intéressons dans ce travail qu'aux techniques basées sur la redondance d'informations, donc qu'aux techniques de compensation des erreurs.

3.2.2.2 Hypothèses de défaillance

La réalisation d'un système tolérant aux fautes est obtenue par l'addition de composants supplémentaires, que ce soit matériels ou logiciels, à ce système durant sa conception. L'algorithme de tolérance aux fautes est l'un de ces composants, il gère les composants restant afin d'assurer le bon fonctionnement du système en présence de défaillances. Puisque les hypothèses d'occurrence des fautes diffèrent d'un système à un autre, le type et le niveau de la redondance introduite dans un système dépend directement de ces hypothèses [48, 63]. Les hypothèses de défaillance des systèmes les plus utilisées dans la littérature [13, 50] sont (figure 3.2) :

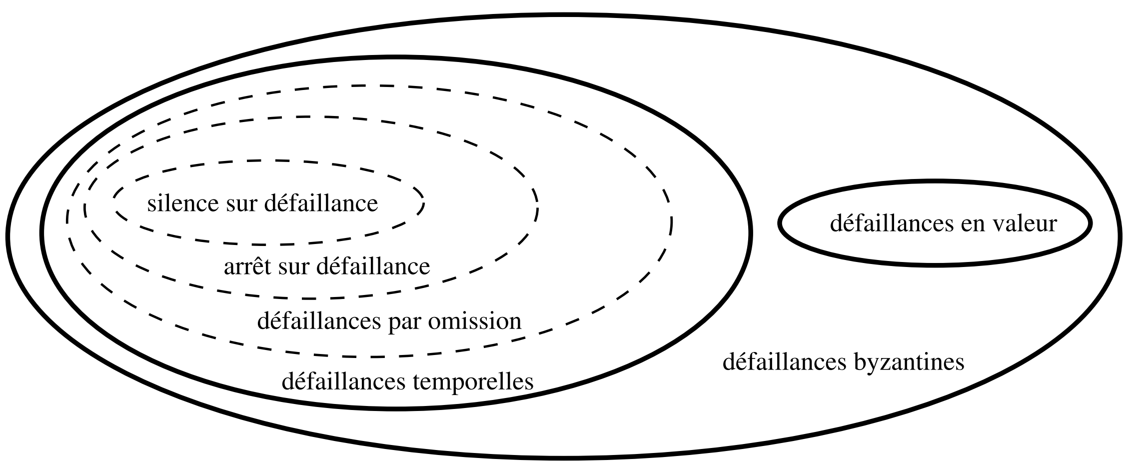


FIG. 3.2 – Couverture entre les hypothèses de défaillance

- **Systèmes à défaillances en valeur** : ces systèmes supposent que les valeurs sont délivrées à temps et qu'ils défaillent uniquement si les valeurs délivrées sont fausses,

- **Systèmes à silence sur défaillances** : un système à silence sur défaillances soit fonctionne correctement (valeurs correctes et délivrées à temps), soit est défaillant et il arrête alors de fonctionner. MARS est un exemple de ce type de systèmes [68] ;
- **Systèmes à arrêt sur défaillances** : ce sont des systèmes à silence sur défaillances, mais, avant que le système arrête son fonctionnement, il délivre un message aux autres systèmes indiquant son arrêt. Un exemple de ces systèmes est le *processeur à arrêt sur défaillance* [71]. Ce processeur est un composant physique, qui en présence de fautes, spécifiées dans ses hypothèses de défaillances, génère un message indiquant son arrêt ;
- **Systèmes à défaillances par omission** : ces systèmes supposent que les valeurs sont correctes, et qu'ils défaillent uniquement si la valeur n'est jamais délivrée. Par exemple, un système perd des messages sortants (omission en émission) ;
- **Systèmes à défaillances temporelle** : ces systèmes supposent que les valeurs délivrées sont correctes, et qu'ils défaillent uniquement si la valeur est temporellement délivrée trop tôt (en avance par rapport à sa plage de temps spécifiée) ou trop tard (en retard par rapport à sa plage de temps spécifiée),
- **Systèmes à défaillances byzantines** : ces systèmes défaillent d'une manière arbitraire. Ce mode de défaillance est parfois considéré par des systèmes à très haute fiabilité (nucléaire, spatial).

Enfin, notons qu'une hypothèse de défaillance peut couvrir une ou plusieurs autres hypothèses. La couverture entre les hypothèses que nous avons citées est illustrée sur la figure 3.2.

3.2.2.3 Tolérance aux fautes matérielles et logicielles

La défaillance d'un système peut être la conséquence d'une faute matérielle ou d'une faute logicielle.

A. Tolérance aux fautes logicielles

Les fautes logicielles sont dues aux erreurs de conception des composants logiciels de l'algorithme. Un exemple très intéressant de faute logicielle est l'explosion de la fusée Ariane 5 en juin 1996 à cause des erreurs de conception du logiciel [4]. L'*uni-version du logiciel* et la *multi-version du logiciel* [41, 79] sont les deux techniques de base les plus utilisées pour tolérer des fautes logicielles d'un système.

- *Uni-version du logiciel* : le but principal des approches utilisant cette technique est de tolérer les fautes logicielles en utilisant une seule version du logiciel, ou d'un de ses composants. Pour tolérer la faute de chaque uni-version du logiciel, le concepteur du système doit la modifier en lui ajoutant des mécanismes de détection et de traitement d'erreurs. Parmi les approches utilisant cette technique, on trouve : le traitement d'exceptions, la détection d'erreur, le point de reprise (check-point and restart).
- *Multi-version du logiciel* : la multi-version du logiciel est une technique de tolérance aux fautes logicielles basée sur le principe de la redondance logicielle, où chaque composant logiciel

est répliqué en plusieurs versions différentes. L'avantage de cette technique par rapport à la première technique est que ces versions logicielles peuvent être exécutées en séquence ou en parallèle pour tolérer les fautes de certaines versions. En plus, les algorithmes implantant ces versions logicielles peuvent être, tous ou certains, développés par différents concepteurs sur différents outils. Parmi les approches utilisant cette technique, on trouve : N-version de programmation. C'est le cas des commandes de vol des avions Airbus et Boeing [80, 84].

B. Tolérance aux fautes matérielles

Les fautes matérielles sont souvent dues soit aux fautes de conception et de fabrication des composants matériels du système, soit à l'interaction du système avec l'environnement qu'il contrôle. Elles peuvent être tolérées soit en utilisant des solutions logicielles, soit en utilisant des solutions matérielles. Les solutions logicielles sont basées sur la redondance des composants logiciels, et les solutions matérielles sont basées sur la redondance des composants matériels. Nous ne nous intéressons dans ce travail qu'aux solutions logicielles que nous détaillerons dans le chapitre 5.

Hypothèse 1 *Dans ce travail, nous ne nous intéressons qu'aux fautes matérielles, et nous supposons que l'algorithme est fiable et sans fautes, c'est-à-dire que le logiciel a été validé par des méthodes de tolérance aux fautes logicielles [41, 79], tels que le model-checking, le theorem-proving, le traitement des exceptions …*

Dans la suite de ce mémoire nous désignons par « faute » une « faute matérielle », et nous ne parlons que des algorithmes de tolérance aux fautes matérielles basées sur la redondance logicielle.

3.2.3 Problème de distribution et d'ordonnancement temps réel et tolérant aux fautes

La co-existence d'un algorithme de distribution/ordonnancement et d'un algorithme de tolérance aux fautes dans un système réactif nécessite la mise en œuvre des mécanismes de communication et de synchronisation entre ces deux algorithmes, puisque lors de la détection d'un état erroné dans le système, l'algorithme de tolérance aux fautes doit choisir le type et le niveau de la redondance logicielle à introduire dans le système, ensuite l'algorithme de distribution/ordonnancement se charge de l'allocation spatiale et temporelle des composants logiciels redondants sur les composants matériels de l'architecture, tout en respectant des contraintes temporelles. Cependant, ces mécanismes de communication et de synchronisation sont coûteux et difficiles à réaliser. Donc, la plupart des systèmes combinent ces deux algorithmes en un seul algorithme, appelé *algorithme de distribution/ordonnancement tolérant aux fautes*.

Le but d'un tel algorithme est de résoudre le problème de la recherche d'une allocation valide des composants logiciels de l'algorithme sur les composants matériels de l'architecture, tout en tolérant des fautes matérielles. Ce problème peut être formalisé comme suit :

Problème 2 *Étant donnés :*

- *une architecture matérielle hétérogène $\mathcal{A}rc$ composée de n composants matériels :*

$$\mathcal{A}rc = \{p_1, ..., p_n\}$$

- *un algorithme $\mathcal{A}lg$ composé de m composants logiciels :*

$$\mathcal{A}lg = \{o_1, ..., o_m\}$$

- *des caractéristiques d'exécution $\mathcal{E}xe$ des composants de $\mathcal{A}lg$ sur les composants de $\mathcal{A}rc$,*
- *un ensemble de contraintes matérielles $\mathcal{D}is$ et un ensemble de contraintes temps réel $\mathcal{R}tc$,*
- *un sous-ensemble $\mathcal{F}$ de $\mathcal{A}rc$ de composants matériels qui peuvent causer la défaillance du système,*
- *un ensemble de critères à optimiser,*

il s'agit de trouver une application $\mathcal{A}$ qui réplique un ou plusieurs composants o_i de $\mathcal{A}lg$, place chaque composant répliqué o_i^j sur un composant p_j de $\mathcal{A}rc$, et lui assigne un ordre d'exécution t_k sur son processeur :

$$\begin{array}{rccl} \mathcal{A}: & \mathcal{A}lg & \longrightarrow & \mathcal{A}rc \\ & o_i & \longmapsto & \mathcal{A}(o_i^j) = (p_j, t_k) \end{array}$$

qui doit satisfaire $\mathcal{D}is$ et $\mathcal{R}tc$, tolérer les fautes des composants de $\mathcal{F}$, et optimiser les critères spécifiés.

3.3 Fiabilité des systèmes réactifs

3.3.1 Terminologies

Plusieurs définitions du concept fiabilité ont été données dans la littérature [13, 49, 50, 56]. La définition suivante est donnée par Lakey et Neufelder dans [49] :

Définition 21 (Fiabilité) *La fiabilité est la probabilité qu'un système, y compris tous ces composants matériels et logiciels, accomplisse sa mission dans un intervalle de temps déterminé et dans des conditions d'environnement déterminées.*

Formellement, la fiabilité est définie par une fonction de temps, notée $\mathcal{F}(t)$, qui exprime la probabilité qu'un système fonctionne correctement durant une période de temps définie par l'intervalle $]t_0, t]$, sachant que ce système fonctionnait correctement à l'instant t_0.

Si on considère que T est la durée de vie d'un composant matériel (son temps de bon fonctionnement avant sa défaillance), et que $f(t)$ est la densité de probabilité de défaillance de T, alors la probabilité qu'un système défaille à l'instant t est donnée par la fonction $\overline{\mathcal{F}}(t)$, définie par :

$$\overline{\mathcal{F}}(t) = P(T \leq t) = \int_{t_0}^{t} f(x)dx \tag{3.1}$$

Étant donné qu'un système est soit fonctionnel soit ne l'est pas, sa probabilité de bon fonctionnement (ou sa fiabilité) peut donc être calculée en utilisant l'équation (3.1) par la formule suivante :

$$\mathcal{F}(t) = 1 - \overline{\mathcal{F}}(t) = 1 - P(T \leq t)$$

Dans le cas d'un système complexe, le calcul de la probabilité de défaillance d'un système dépend d'une mesure importante, qui est le *taux de défaillance* de ses composants logiciels et matériels.

Définition 22 (Taux de défaillance) *Le taux de défaillance d'un composant est une fonction du temps, notée $\lambda(t)$, qui exprime le nombre de défaillances de ce composant dans un intervalle de temps déterminé.*

La loi exponentielle est la loi la plus utilisée pour représenter la probabilité qu'un système défaille à un instant t donné [13]. Elle suppose que le taux de défaillance de chaque composant est une *constante* strictement positive ($\lambda(t) = \lambda$). Donc, pour tout $t \geq 0$:

$$\overline{\mathcal{F}}(t) = 1 - e^{-\lambda t}$$

ce qui donne :

$$\mathcal{F}(t) = e^{-\lambda t}$$

Dans certains modèles, le taux de défaillance est représenté par une autre mesure qui est le temps moyen jusqu'à l'occurrence de la première défaillance, noté MTTF « Mean Time To Failure ». La relation entre $\lambda(t)$ et MTTF est donnée par la formule :

$$MTTF = \frac{1}{\lambda(t)}$$

3.3.2 Modèles pour le calcul de la fiabilité

Le calcul de la fiabilité d'un système nécessite la définition d'un modèle de fiabilité. Le choix d'un tel modèle de fiabilité a un effet direct sur le niveau de confiance de ce calcul. Ce modèle doit définir les paramètres de performance des composants logiciels et/ou matériels du système (tels que les taux de défaillance), le niveau et le type de la redondance si elle existe, ainsi que les hypothèses de défaillance. Il existe dans la littérature plusieurs modèles de fiabilité. Les modèles les plus utilisés sont : les modèles combinatoires [1], les modèles basés sur les chaînes de Markov [70], les modèles basés sur les réseaux de Petri [59] et les modèles basés sur la théorie de l'ordonnancement [73]. Nous les présentons dans les paragraphes suivants.

3.3.2.1 Modèles combinatoires

Généralement, ces modèles utilisent la théorie des graphes pour représenter graphiquement (par un graphe orienté) toutes les combinaisons d'événements élémentaires qui peuvent causer la défaillance d'un système. À chaque nœud sans prédécesseur du graphe, qui représente un événement élémentaire, est associé un ensemble de paramètres de performance, telle que la probabilité de son apparition. La fiabilité de ce système est calculée à partir d'une analyse quantitative de chaque graphe. Les deux modèles les plus fréquemment utilisés sont le diagramme de blocs et l'arbre de fautes. Par exemple, l'arbre de fautes est constitué de plusieurs niveaux, où chaque nœud d'un niveau supérieur représente une combinaison de deux ou plusieurs événements liés aux nœuds de niveau inférieur. Les feuilles de l'arbre représentent les événements élémentaires qui peuvent causer la défaillance d'un système, et la racine de l'arbre représente l'événement de défaillance du système. Donc, la probabilité que le système défaille est la probabilité d'atteindre la racine de l'arbre à partir de ses feuilles. Les modèles combinatoires sont faciles à comprendre, mais il n'est pas facile de représenter le comportement non indépendant des événements, au sens probabiliste.

3.3.2.2 Modèles basés sur les chaînes de Markov

Les chaînes de Markov permettent de modéliser le comportement dynamique d'un système par un graphe d'états, qui représente tous les états du système et les transitions possibles entre ces états. Les transitions sont pondérées par des probabilités suivant des lois exponentielles. Le calcul de la fiabilité d'un système peut être effectué grâce à des méthodes de résolution numérique ou par simulation. À la différence des modèles combinatoires les chaînes de Markov permettent la modélisation des événements non indépendants et aussi des événements de réparation des composants du système. Cependant, l'espace d'états peut grossir exponentiellement avec le nombre de composants d'un système, d'où des problèmes algorithmiques pour calculer la fiabilité.

3.3.2.3 Modèles basés sur les réseaux de Petri

Le comportement dynamique d'un système est ici représenté par un ensemble d'états, de jetons et de transitions. À la différence des modèles combinatoires, les transitions peuvent être associées à n'importe quel type de loi probabiliste. Les réseaux de Petri peuvent être utilisés pour générer des chaînes de Markov. En plus, ils peuvent être utilisés facilement pour représenter les caractéristiques des systèmes concurrents, telles que la synchronisation et le partage des ressources, et aussi pour valider des propriétés d'un système, telle que l'absence de blocage. Le calcul de la fiabilité est basé sur la simulation [24]. Le but de la simulation est d'appliquer à un système un ensemble de tests aléatoires, et d'utiliser ensuite les résultats de ces tests pour calculer la fiabilité de ce système. Cependant, la précision de ce calcul dépend du choix de l'ensemble de tests et augmente avec la durée de la simulation. Or la procédure de production des tests introduit toujours un biais, qui est difficile à mesurer.

3.3.2.4 Modèles basés sur la théorie de l'ordonnancement

Plusieurs travaux [16, 40, 45, 46, 54, 55, 65, 76] ont montré que l'utilisation de la théorie de l'ordonnancement pour la conception des systèmes peut améliorer la fiabilité de ces systèmes, c'est pourquoi nous nous sommes attachés à ce modèle. Dans ce modèle, un taux de défaillance est associé à chaque composant matériel, et le calcul de la fiabilité est basé sur une fonction d'évaluation ($\mathcal{F}iab$), qui permet d'évaluer la probabilité du bon fonctionnement du système (ou de l'allocation résultante d'un algorithme de distribution/ordonnancement). La fiabilité d'un système dépend donc de l'allocation des composants logiciels de l'algorithme sur les composants matériels de l'architecture. La conséquence de l'utilisation d'une telle fonction d'évaluation est que l'algorithme de distribution/ordonnancement doit prendre en compte les taux de défaillance des composants matériels durant sa phase d'allocation spatiale des composants logiciels sur les composants matériels.

3.3.3 Problème de distribution et d'ordonnancement temps réel et fiable

Durant le processus de conception d'un système réactif, le but d'un algorithme de distribution/ordonnancement est d'allouer spatialement et temporellement les composants logiciels sur les composants matériels de l'architecture, tout en respectant des contraintes temporelles et matérielles. Cependant, cette allocation ne prend pas en compte la probabilité de défaillance des composants matériels, ce qui peut avoir un effet considérable sur l'augmentation de la probabilité de défaillance d'un système [73]. Donc, il est important que les algorithmes de distribution/ordonnancement prennent en compte les taux de défaillance des composants matériels durant leur processus d'allocation spatiale. Ceci afin de réduire l'effet des défaillances sur l'exécution des composants logiciels, d'où l'augmentation de la probabilité de bon fonctionnement du système.

Nous avons ainsi un nouveau problème de distribution/ordonnancement que nous appelons « *problème de distribution/ordonnancement temps réel et fiable* ». Il peut être formalisé comme suit :

Problème 3 *Étant donnés :*

- *une architecture matérielle hétérogène $\mathcal{A}rc$ composée de n composants matériels :*

$$\mathcal{A}rc = \{p_1, ..., p_n\}$$

- *un algorithme $\mathcal{A}lg$ composée de m composants logiciels :*

$$\mathcal{A}lg = \{o_1, ..., o_m\}$$

- *des caractéristiques d'exécution $\mathcal{E}xe$ des composants de $\mathcal{A}lg$ sur les composants de $\mathcal{A}rc$,*
- *un ensemble de contraintes matérielles $\mathcal{D}is$ et un ensemble de contraintes temps réel $\mathcal{R}tc$,*
- *un ensemble de taux de défaillances $\mathcal{T}_{def}$ des composants matériels,*
- *un ensemble de critères à optimiser, parmi lesquels une fonction $\mathcal{F}iab$ d'évaluation de la fiabilité de ce système,*

il s'agit de trouver une application $\mathcal{A}$ qui place chaque composant o_i de $\mathcal{A}lg$ sur un composant p_j de $\mathcal{A}rc$, et qui lui assigne un ordre d'exécution t_k sur son processeur :

$$\begin{array}{rlcl} \mathcal{A}: & \mathcal{A}lg & \longrightarrow & \mathcal{A}rc \\ & o_i & \longmapsto & \mathcal{A}(o_i) = (p_j, t_k) \end{array}$$

qui doit satisfaire $\mathcal{D}is$ et $\mathcal{R}tc$, et optimiser les critères spécifiés, parmi lesquels maximiser la fonction $\mathcal{F}iab$.

Remarque 4 *L'architecture matérielle est hétérogène, donc les taux de défaillance de ses divers composants (processeurs, média de communication ...) sont a priori distincts.*

3.4 Conclusion

Dans ce chapitre, nous avons présenté deux aspects différents de la sûreté de fonctionnement des systèmes réactifs distribués et embarqués, qui sont la tolérance aux fautes et la fiabilité. La tolérance aux fautes est mise en œuvre par la détection des erreurs et le traitement des erreurs. La fiabilité représente une mesure quantitative qui permet d'évaluer le bon fonctionnement d'un système durant son exploitation.

Dans ce travail, nous utiliserons la technique de la compensation des erreurs pour tolérer des fautes matérielles dans un système distribué réactif embarqué. Afin d'évaluer la fiabilité de ces systèmes, nous utiliserons un modèle basé sur la théorie de l'ordonnancement. Ces choix sont motivés par les caractéristiques des systèmes considérés et les contraintes auxquelles ils sont soumis : embarquabilité, temps réel, distribution et criticité.

Deuxième partie

Méthodologies pour la génération de distributions et d'ordonnancements temps réel, fiables et tolérants aux fautes

« Le difficile, ce n'est pas de donner,
c'est de ne pas tout donner. »
Sidonie GABRIELLE

Chapitre 4

Modèles

Résumé

Ce chapitre présente les modèles prisent en compte dans ce travail. Ce sont le modèle d'algorithme, le modèle d'architecture et le modèle d'exécution. Le modèle d'algorithme décrit la spécification des composants logiciels de l'algorithme. Le modèle d'architecture décrit la spécification de l'architecture matérielle distribuée. Et le modèle d'exécution décrit le mode d'exécution des composants logiciels de l'algorithme sur les composants matériels de l'architecture.

4.1 Introduction

Dans la spécification des systèmes informatiques, on peut distinguer deux approches principales de spécification : une approche non formelle et une approche formelle. L'approche non formelle, appelée approche classique, consiste à écrire à la main une spécification papier, et à partir de cette spécification l'implantation peut être écrite à la main dans un langage de programmation. L'approche formelle consiste à écrire la spécification dans un langage de spécification de haut niveau, ce qui permet en premier temps la vérification de cette spécification par des outils de validation formelle (vérification, preuve, ...), et en deuxième temps à utiliser des outils de génération automatique de code pour produire du code compilable (par exemple du C).

En raison de la criticité des domaines applicatifs considérés (nucléaire, avionique, spatial, ...) nous avons opté pour l'approche formelle pour la spécification des systèmes distribués réactifs embarqués. Ces systèmes peuvent être décomposés en deux parties principales, qui sont l'algorithme et l'architecture matérielle. La spécification de ces systèmes consiste à décrire l'algorithme (modèle d'algorithme), l'architecture matérielle (modèle d'architecture), et le mode d'exécution de l'algorithme sur l'architecture matérielle (modèle d'exécution). Nous donnons dans ce chapitre une brève présentation de ces trois modèles qui sont basés sur les mêmes modèles que la méthodologie AAA[1] [35, 74] implantée dans SYNDEX. AAA consiste à mettre en correspondance de manière efficace l'algorithme sur l'architecture matérielle pour réaliser une implantation optimisée, elle est basée sur l'heuristique de distribution/ordonnancement temps réel que nous avons présentée dans le chapitre 2 (cf. section 2.6, page 27).

4.2 Modèle d'algorithme

Le logiciel d'un système réactif peut être décomposé en deux grandes parties :

- l'**algorithme applicatif -** il décrit le comportement du système. Il peut être décomposé en plusieurs sous-algorithmes applicatifs, où chaque sous-algorithme réalise une fonction précise,
- l'**exécutif -** son rôle consiste à gérer et à allouer les ressources physiques à l'algorithme applicatif, et aussi à respecter les contraintes temporelles.

Nous ne nous intéressons dans ce chapitre qu'à la modélisation de l'algorithme applicatif du logiciel. Le modèle d'algorithme que nous présentons dans ce chapitre est basé sur celui développé dans la thèse d'Annie Vicard [81].

4.2.1 Hypergraphe orienté

L'algorithme est modélisé par un graphe flot de données, qui est un hypergraphe orienté, appelé *graphe d'algorithme*. Les sommets de l'hypergraphe désignent les *opérations*, et les arcs désignent les *dépendances de données* entre opérations. Ces arcs induisent un ordre partiel d'exécution sur les opérations, qui décrit le parallélisme potentiel (opérations non reliées par une dépendance de donnée) disponible au niveau de l'algorithme entre les opérations.

Définition 23 (Opération) *Une opération est une entité logicielle qui représente une séquence finie de code informatique.*

Remarque 5 *Dans certains systèmes réactifs le nom opération désigne une tâche temps réel. Le terme opération a été choisi ici pour désigner un lien fonctionnel avec la notion d'opérateur que nous utiliserons dans la modélisation de l'architecture (voir section 4.3).*

Définition 24 (Dépendance de donnée) *Une dépendance de donnée* $(o_i \triangleright o_j)$ *représente la relation de précédence (transfert de données) entre l'opération* o_i *et l'opération* o_j*. L'opération* o_i *est appelée prédécesseur de* o_j*, et l'opération* o_j *est appelée successeur de* o_i*. Elle est appelée dépendance de donnée de diffusion si elle possède une seule opération productrice et plusieurs opérations consommatrices. Dans ce cas, elle est notée* $\big(o_i \triangleright \{\ldots, o_j, \ldots\}\big)$.

On peut distinguer dans un graphe d'algorithme deux types d'opérations : opérations de calcul et opérations d'entrée/sortie.

[1] AAA : Aéquation Algorithme Architecture

Définition 25 (Opérations de calcul) *Une opération de calcul possède au moins un prédécesseur et au moins un successeur. Elle consomme des données d'entrée qui proviennent de ses prédécesseurs, et produit des données de sortie qui sont ensuite utilisées par ses successeurs. Elle ne peut s'exécuter que lorsque toutes ses données d'entrée sont présentes.*

Définition 26 (Opérations d'entrée/sortie) *Une opération d'entrée (resp. de sortie) s'effectue sur une interface d'entrée, capteur (resp. de sortie, actionneur) avec l'environnement extérieur. Les opérations d'entrée (resp. de sortie) sont des opérations sans prédécesseur (resp. sans successeur). Le rôle d'une opération d'entrée est de transformer les informations physiques issues d'un capteur en données numériques qui sont ensuite consommées par une ou plusieurs opérations de calcul. Une opération de sortie transforme les données numériques issues des opérations de calcul en grandeurs physiques qui sont ensuite appliquées aux actionneurs.*

La figure 4.1 est un exemple d'un graphe d'algorithme composé de deux opérations d'entrées In_1 et In_2, quatre opérations de calcul A, B, C et D, une opération de sortie Out_1, six dépendances de données $\big(In_2 \triangleright B\big)$, $\big(In_2 \triangleright C\big)$, $\big(A \triangleright D\big)$, $\big(B \triangleright D\big)$, $\big(C \triangleright Out_1\big)$ et $\big(D \triangleright Out_1\big)$, et d'une dépendance de diffusion $\big(In_1 \triangleright \{A, B\}\big)$. Ici, par exemple, les opérations A, B et C peuvent être exécutées en parallèle sur trois processeurs distincts (parallélisme potentiel).

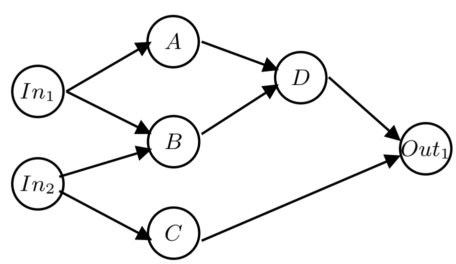


FIG. 4.1 – Exemple d'un graphe d'algorithme.

4.2.2 Hypergraphe orienté infiniment répété

Dans un système réactif, l'algorithme est en interaction *infiniment répétée* avec son environnement. Chaque interaction se décompose généralement en trois phases complémentaires : l'acquisition, le calcul et l'émission. Son mode de fonctionnement est le suivant :

```
A chaque top faire
  l'acquisition des données issues des capteurs;
  le calcul de la loi de commande;
  l'émission des commandes aux actionneurs;
fin faire
```

Ici, `top` est une horloge dont le rythme doit être fixé en fonction des caractéristiques de l'environnement, c'est-à-dire du rythme d'évolution des grandeurs physiques.

Donc, la séquence « acquisition-calcul-émission » est infiniment exécutée, et chaque exécution définit une *répétition* du graphe d'algorithme. Afin de prendre en compte ce mode infiniment répété de l'algorithme dans notre modèle, l'algorithme est modélisé par un hypergraphe acyclique (figure 4.2), constitué d'un *motif* infiniment répété [81].

Définition 27 (Motif) *On appelle motif d'un graphe, l'ensemble des opérations appartenant à la même répétition et l'ensemble des dépendances de données entre ces opérations.*

Ainsi, dans certaines applications temps réel, une opération a besoin lors de sa n-ième exécution (c'est-à-dire lors de la n-ième répétition du graphe d'algorithme) de consommer une donnée produite par une autre opération lors de la (n-1)-ième répétition, d'où la notion de *dépendances de données inter-répétition* (ou inter-motif).

Par exemple, la figure 4.2 est l'hypergraphe infiniment répété du graphe d'algorithme de la figure 4.1. On peut distinguer dans ce graphe plusieurs motifs identiques et plusieurs instances de la dépendance de donnée inter-motif $\left(A^{(n-1)} \triangleright B^{(n)}\right)$.

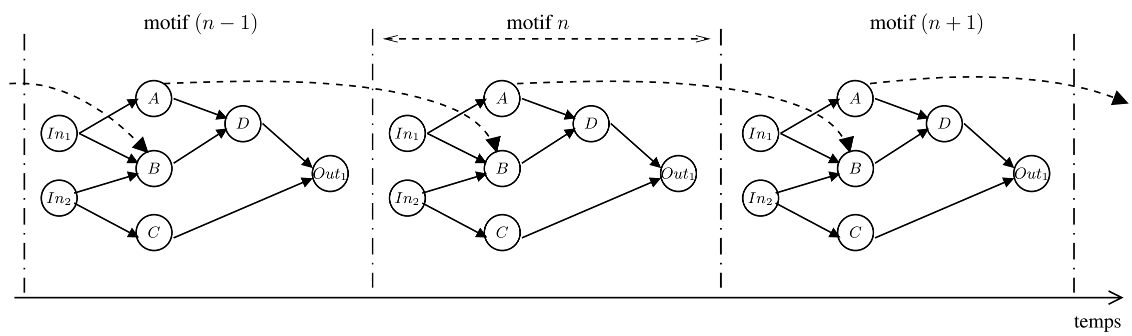


FIG. 4.2 – Exemple d'un graphe d'algorithme infiniment répété.

Cependant, ce modèle pose un problème qui est la taille importante de la spécification de l' algorithme. Afin de réduire cette taille, le graphe d'algorithme est factorisé [52]. La factorisation consiste à superposer toutes les instances du graphe d'algorithme ; elle est réalisée en deux temps. Dans un premier temps, l'algorithme est modélisé par un hypergraphe factorisé constitué d'un seul motif de l'hypergraphe infiniment répété. Dans un deuxième temps, toutes les instances d'une même dépendance de donnée inter-motif $\left(A^{(n-1)} \triangleright B^{(n)}\right)$ sont remplacées par une nouvelle opération factorisée, appelée *opération mémoire* (ou *retard* [75]) M_{AB}, et deux dépendances intra-motif factorisées $\left(A \triangleright M_{AB}\right)$ et $\left(M_{AB} \triangleright B\right)$. Une opération mémoire introduit dans notre modèle la notion d'*état*, et donc l'ensemble des opérations mémoires représente l'état de l'algorithme.

Définition 28 (Opération mémoire) *Elle est appelée mémoire puisque elle mémorise la donnée produite par une opération A lors de son exécution précédente à la répétition $n-1$. Cette donnée sera ensuite consommée par son successeur B à la répétition n. Nous supposons que sa durée d'exécution est nulle.*

« Notre modèle suit la sémantique des langages synchrones. Cela veut dire, que pour assurer un parfait déterminisme nécessaire dans le contexte temps réel, d'une part on fait l'hypothèse que lorsqu'une opération reçoit ses données, elle s'exécute et produit instantanément ses données, cela étant vrai pour toute opération du graphe, celui-ci s'exécute donc instantanément, et d'autre part afin d'assurer qu'un motif sera terminé avant qu'un autre commence, on introduit un arc de précédence sans donnée entre chaque motif. Les exécutions successives motif définissent un temps logique dont chaque instant correspond à une réaction de l'application. » [75]

Notons qu'une donnée initiale pour chaque opération mémoire est nécessaire lors de la première répétition. Donc, nous ajoutons dans ce nouveau graphe pour chaque opération mémoire une nouvelle opération, appelée *opération constante*, et une nouvelle dépendance de données entre cette nouvelle opération et l'opération mémoire.

Définition 29 (Opération constante) *Elle est appelée constante puisque sa valeur ne change pas durant l'exécution du système. Elle est exécuté une seule fois pendant la première répétition, et nous supposons que sa durée d'exécution est nulle.*

Par exemple, la figure 4.3 est le graphe d'algorithme factorisé du graphe d'algorithme infiniment répété de la figure 4.2. Dans ce graphe factorisé, chaque opération (resp. dépendance de donnée intra et inter-motif) représente plusieurs instances d'une même opération (resp. dépendance de donnée intra et inter-motif).

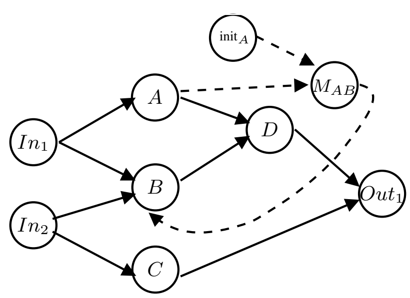


FIG. 4.3 – Exemple d'un graphe d'algorithme factorisé.

Enfin, l'algorithme d'une application temps réel est formellement défini par :

Définition 30 (Algorithme) *Un algorithme est modélisé par un hypergraphe factorisé, appelé graphe d'algorithme $\mathcal{A}lg$. Il est représenté par le couple (O,E), où O est l'ensemble des n opérations et E est l'ensemble des m dépendances de données.*

Par exemple, pour le graphe d'algorithme de la figure 4.3, nous avons :

$$\begin{array}{rcl} O & = & \{In_1, In_2, A, B, C, D, M_{AB}, init_A, Out_1\} \\ E & = & \{(In_2 \triangleright B), (In_2 \triangleright C), (A \triangleright D), (B \triangleright D), (C \triangleright Out_1), (D \triangleright Out_1), \\ & & (In_1 \triangleright \{A, B\}), (init_A \triangleright M_{AB}), (A \triangleright M_{AB}), (M_{AB} \triangleright B)\} \end{array}$$

4.3 Modèle d'architecture

Dans plusieurs domaines très variés tels que l'automobile, l'aéronautique, et la production d'énergie, la conception d'une application temps réel critique nécessite soit l'utilisation d'une architecture monoprocesseur offrant une puissance de calcul élevée, soit l'utilisation d'une architecture multiprocesseur parallèle basée sur un modèle de type PRAM[2] [20] ou distribuée basée sur un modèle de type DRAM[3] [20].

Cependant, certaines applications ne peuvent satisfaire les contraintes temps réel imposées par l'environnement avec des architectures monoprocesseur, ceci s'explique par l'exécution séquentielle de toutes les opérations de l'algorithme par l'unique processeur. L'utilisation d'architecture multiprocesseur permet d'exploiter le parallélisme potentiel disponible au niveau de l'algorithme, pour le transformer en parallélisme disponible au niveau de l'architecture ; ceci permet de réduire le temps d'exécution de l'algorithme. Donc, nous nous intéressons dans ce travail aux architectures multiprocesseur, et plus particulièrement aux architectures distribuées puisque elles permettent la délocalisation de certaines fonctionnalités, par exemple pour se rapprocher des capteurs et des actionneurs.

Avant de présenter le modèle d'architecture, il est nécessaire d'identifier et de caractériser tous les composants de l'architecture, ainsi que de connaître la topologie du réseau de communication. Généralement une architecture distribuée peut être constituée de composants programmables (processeurs) et de composants spécialisés (ASIC ou FPGA). Dans ce travail, nous nous intéressons aux architectures constituées uniquement de processeurs. Plusieurs définitions du terme processeur ont été données dans la littérature. La définition suivante est basée sur le modèle de Von Neumann :

Définition 31 (Processeur) *Un processeur est une machine à états finis, composé de quatre unités : une unité arithmétique et logique (UAL), une unité d'entrée/sortie (ES), une unité de contrôle (UC), et une mémoire locale (RAM [4]).*

Donc, l'architecture distribuée que nous modélisons possède les caractéristiques suivantes :

- elle est constituée d'un ensemble de machines monoprocesseur reliées entre elles par un réseau de communication,
- les processeurs peuvent avoir des caractéristiques identiques (c'est le cas pour les architectures homogènes), ou des caractéristiques différentes (c'est le cas pour les architectures hétérogènes),
- les processeurs peuvent communiquer entre eux par *passage de messages* sur un réseau de communication composé d'un ou plusieurs média de communication ; chaque médium de communication peut relier deux ou plusieurs processeurs et est constitué d'une ou plusieurs liaisons physiques et d'une mémoire SAM[5] qui fonctionne en mode FIFO.

[2] PRAM = Parallel Random Access Machine.
[3] DRAM = Distributed Random Access Machine.
[4] RAM = Random Access Memory.
[5] SAM = Sequential Access Memory

La figure 4.4 représente un exemple d'une architecture distribuée constituée de trois processeurs P_1, P_2 et P_3, et d'un réseau de communication composé de deux média de communication basés sur deux mémoires SAM.

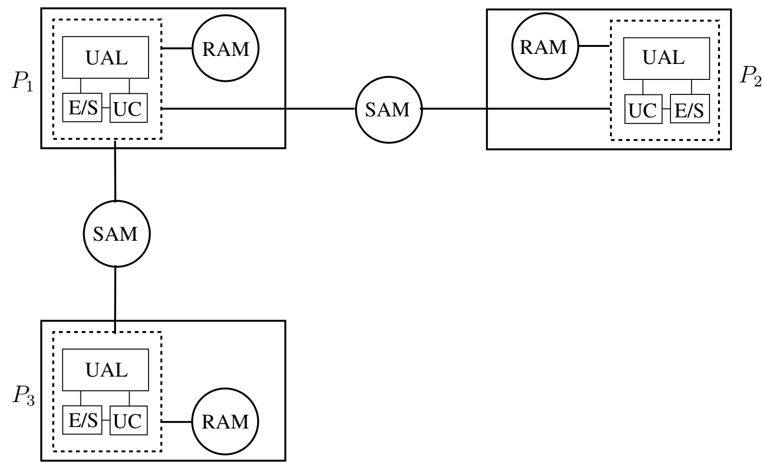


FIG. 4.4 – Exemple d'une architecture distribuée.

Cependant, ce type d'architecture n'offre pas de réel parallélisme entre les opérations de calcul et les opérations de communication. Ceci est du au temps passé par un processeur pour effectuer un transfert de données entre la mémoire SAM et sa mémoire RAM lors d'une opération de communication : le processeur interrompt son opération de calcul en cours pour exécuter ce transfert. Afin de résoudre ce problème de parallélisme entre opérations de calcul et opérations de communication au niveau du processeur, parfois les processeurs sont dotés d'une *unité de communication*, appelée DMA[6]. Son rôle est de libérer le processeur de cette tâche de transfert de données entre les mémoires SAM et RAM, et ainsi des opérations de communication. Elle est souvent composée de plusieurs canaux de communication que nous appelons *communicateurs*.

Définition 32 (Communicateur) *Un communicateur est une machine à états finis. Il permet le transfert de données entre deux ou plusieurs processeurs. Il est connecté à une mémoire locale RAM et une mémoire partagée SAM.*

Donc, un processeur est identifié par ses cinq unités : UAL, ES, UC, RAM, et d'un DMA composé d'un ou plusieurs communicateurs. Par exemple, la figure 4.5 représente la nouvelle architecture de la figure 4.4. Elle est constituée de trois processeurs et d'un DMA par processeur. La DMA de chaque processeur est composé de deux communicateurs.

Une mémoire SAM peut être de deux types différents : SAM point-à-point et SAM multipoint. Une mémoire SAM point-à-point ne peut être connectée qu'à deux processeurs (figure 4.5), tandis qu'une mémoire SAM multipoint peut être connectée à plus de deux processeurs (figure 4.6).

[6]DMA = Direct Memory Access.

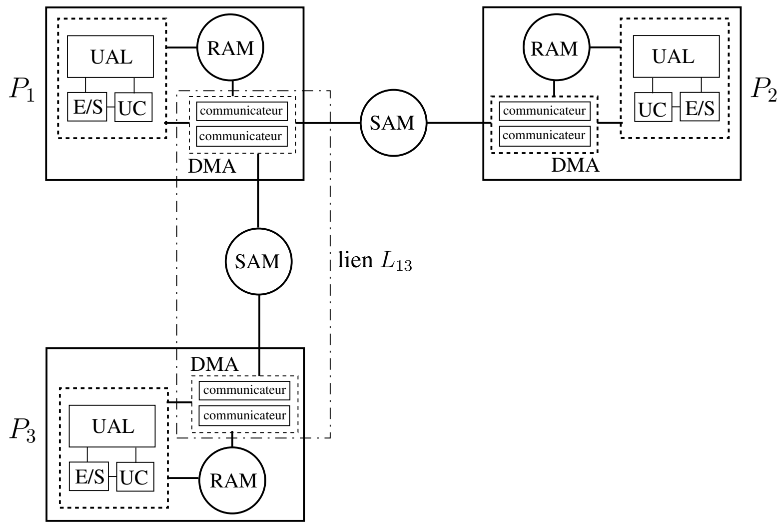


FIG. 4.5 – Exemple d'une architecture distribuée à liaisons point-à-point.

Dans la suite, nous appelons une liaison par mémoire SAM point-à-point un *lien*, et une liaison par mémoire SAM multipoint connectée à tous les processeurs un *bus*. Suivant le type de la mémoire SAM utilisé par le réseau de communication, on peut distinguer deux types d'architectures : architectures à liaisons point-à-point et architectures à liaisons bus.

Définition 33 (Architecture à liaisons point-à-point) *C'est une architecture multiprocesseur distribuée constituée d'un réseau de communication composé uniquement de mémoires SAM point-à-point.*

Définition 34 (Architecture à liaisons bus) *C'est une architecture multiprocesseur distribuée constituée d'un réseau de communication composé uniquement de mémoires SAM multipoint, où chaque mémoire SAM est connectée à tous les processeurs.*

La figure 4.5 est un exemple d'architecture distribuée à liaisons point-à-point. Elle est composée de trois processeurs, et d'un réseau de communication composé de deux liens. Dans cet exemple, chaque DMA de chaque processeur est composé de deux communicateurs. La figure 4.6 est un exemple d'une architecture distribuée à liaisons bus. Elle est composée de trois processeurs, et d'un réseau de communication composé de deux bus.

Remarque 6 *On pourrait considérer un troisième type plus général d'architecture, constituée à la fois de liaisons point-à-point et de liaisons bus. Pour des raisons de simplification, nous ne considérons dans ce travail que les types d'architectures des définitions 33 et 34.*

Afin de modéliser ces deux types d'architectures, plusieurs modèles ont été proposés dans la littérature. Le modèle classique est le modèle le plus utilisé dans le domaine d'ordonnancement. Ce modèle classique modélise l'architecture distribuée par un hypergraphe non orienté, où les

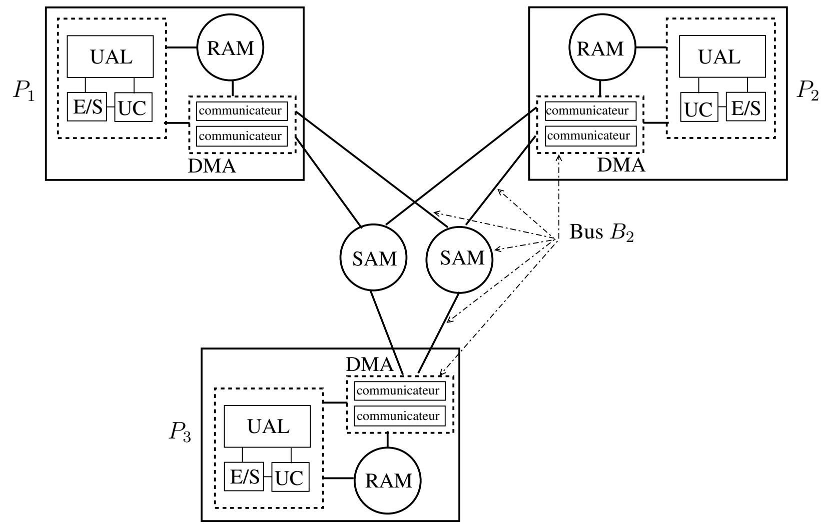


FIG. 4.6 – Exemple d'une architecture distribuée à liaisons bus.

sommets représentent les processeurs, et les hyper-arêtes représentent les liaisons physiques de communication de type point-à-point ou bus. La figure 4.7a (resp. figure 4.7b) représente le modèle classique de l'architecture de la figure 4.5 (resp. figure 4.6).

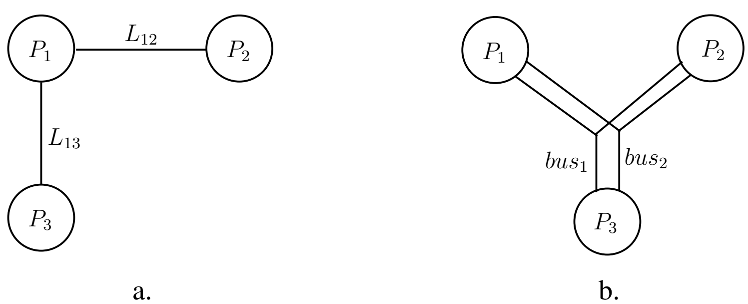


FIG. 4.7 – Modèles classiques.

Étant donné que l'objectif principal de ce travail est la tolérance aux *fautes matérielles*, ce modèle classique ne nous convient pas puisqu'il n'est pas suffisamment fin pour décrire nos modèle de fautes (cf. sections 6.1.1 et 7.1.1). Donc, il est nécessaire de modéliser précisément toutes les ressources physiques de l'architecture distribuée. C'est dans ce but que nous utilisons un modèle d'architecture développé, basé sur celui présenté dans la thèse de Thierry Grandpierre [34].

4.3.1 Architecture à liaisons point-à-point

Dans ce type d'architecture, un processeur est identifié par un *opérateur de calcul* et plusieurs communicateurs.

Définition 35 (Opérateur de calcul) *Un opérateur de calcul est une machine à états finis. Il regroupe l'UAL, l'ES, l'UC, et la mémoire RAM. Son rôle est d'exécuter les opérations de calcul et les opérations d'entrée/sortie du graphe d'algorithme chargées dans sa mémoire RAM.*

L'architecture est modélisée par un graphe non orienté, appelé *graphe d'architecture* $\mathcal{A}rc$. Les sommets du graphe désignent les opérateurs, les communicateurs et les mémoires SAM point-à-point. Ce graphe à deux types d'arêtes, arêtes reliant les opérateurs et les communicateurs d'un même processeur, et arêtes reliant les communicateurs aux mémoires SAM point-à-point.

Par exemple, nous modélisons l'architecture de la figure 4.5 par le graphe de la figure 4.8. Ce graphe est composé de trois opérateurs de calcul OP_1, OP_2 et OP_3, et de deux liens de communication $\{COM_{11}, SAM_{12}, COM_{21}\}$ et $\{COM_{12}, SAM_{13}, COM_{31}\}$.

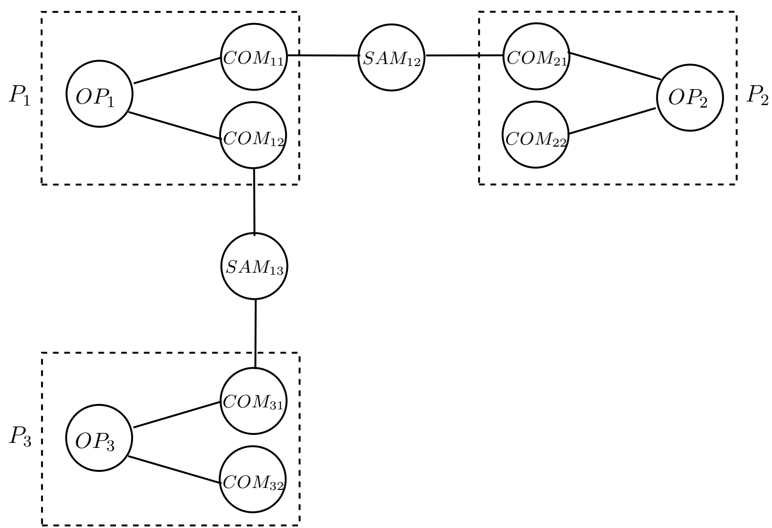


FIG. 4.8 – Modèle d'une architecture à liaisons point-à-point.

4.3.2 Architecture à liaisons bus

Comme pour les architectures à liaisons point-à-point, un processeur est identifié par son opérateur de calcul et ses communicateurs.

L'architecture est modélisée par un graphe non orienté, appelé *graphe d'architecture* $\mathcal{A}rc$. Les sommets du graphe désignent les opérateurs de calcul et les communicateurs, et les mémoires SAM multipoint. Ce graphe à deux types d'arêtes, arêtes reliant les opérateurs de calcul et les communicateurs d'un même processeur, et arêtes reliant les communicateurs aux mémoires SAM multipoint. L'ensemble composé d'un communicateur de chaque processeur et d'une mémoire SAM multipoint reliant ces communicateurs est appelé *bus* de communication.

Par exemple, nous modélisons l'architecture de la figure 4.6 par le graphe de la figure 4.9. Ce graphe est composé de trois opérateurs de calcul OP_1, OP_2 et OP_3, et de deux bus de communication $\{COM_{11}, COM_{22}, COM_{31}, SAM_1\}$ et $\{COM_{12}, COM_{21}, COM_{32}, SAM_2\}$.

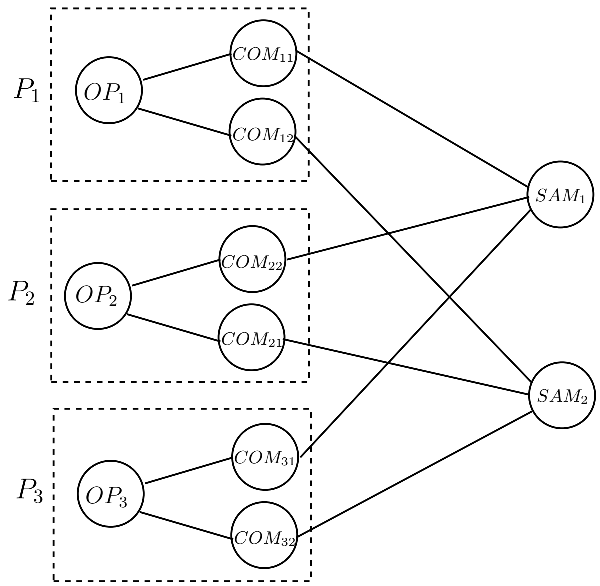


FIG. 4.9 – Modèle d'une architecture à liaisons bus.

Définition 36 (Architecture) *Une architecture $\mathcal{A}rc$ est modélisée par un graphe non orienté, appelé graphe d'architecture. Celui-ci est représenté par le couple (P,M), où P est l'ensemble des n opérateurs de calcul et M est l'ensemble des m média de communication.*

Par exemple, pour le graphe d'architecture de la figure 4.8 :
$P = \{OP_1, OP_2, OP_3\}$,
$M = \{L_{12} = (COM_{11}, SAM_{12}, COM_{21}), L_{13} = (COM_{12}, SAM_{13}, COM_{31})\}$

et pour le graphe d'architecture de la figure 4.9 :
$P = \{OP_1, OP_2, OP_3\}$,
$M = \{B_1 = (COM_{11}, COM_{22}, COM_{31}, SAM_1), B_2 = (COM_{12}, COM_{21}, COM_{32}, SAM_2)\}$

4.4 Modèle d'exécution

Le développement d'une technique de tolérance aux fautes ou de fiabilité est lié au *mode d'exécution* de l'algorithme sur l'architecture, c'est-à-dire à la manière dont est organisée l'exécution de l'algorithme sur l'architecture. Dans la littérature, on peut distinguer deux grandes classes de base pour organiser l'exécution d'un algorithme sur une architecture : *exécution périodique* et *exécution cyclique* [13, 69]. Dans l'exécution périodique, les opérations de l'algorithme sont automatiquement activées à des intervalles réguliers à l'aide d'une horloge périodique externe au système.L'exécution d'une opération parmi toutes les opérations actives est déterminée suivant un ordre de priorité. L'exécution d'une opération peut être preemptée pour exécuter une opération active de plus haute prioritée. Cela est réalisé par l'ordonnanceur dynamique d'un système d'exploitation temps réel résidant à coté du système lui-même. Dans l'exécution cyclique l'exécution des opérations de l'algorithme est répétitive au sein du système lui-même, et une exécution de chaque opération définit une *répétition* d'exécution de l'algorithme sur l'architecture[7]. Cela est réalisé par un ordonnanceur statique. De plus, suivant le type d'application temps réel, les dates de début d'exécution des opérations peuvent être régulières ou non, mais elles sont toutes bornées entre la borne minimale et la borne maximale de la taille d'un cycle.

Dans ce travail, nous ne nous intéressons qu'à l'exécution cyclique de l'algorithme sur l'architecture, et nous décrivons dans cette section la manière suivant laquelle les opérations sont exécutées sur l'architecture. Puisque nous visons des systèmes distribués réactifs embarqués l'exécution des opérations sur l'architecture distribuée est soumise à des contraintes de temps réel et à des contraintes liées à l'architecture (embarquabilité et distribution) que nous présentons également dans cette section.

4.4.1 Exécution cyclique de l'algorithme sur l'architecture

Le mode d'exécution cyclique des composants logiciels (opérations et dépendances de données) de l'algorithme $\mathcal{A}lg$ (par exemple, de la figure 4.3) sur chaque composant matériel de l'architecture $\mathcal{A}rc$ est de la forme suivante :

While true do
 exécuter In_1 ;
 transférer les données de sortie de In_1 vers l'opérateur implantant A ;
 transférer les données de sortie de In_1 vers l'opérateur implantant B ;
 . . .
 . . .
 exécuter Out_1 ;
end While

Remarque 7 *Dans notre modèle d'exécution cyclique de l'algorithme sur l'architecture, nous supposons que chaque opération est exécutée une seule fois durant chaque cycle d'exécution.*

Chaque opération du graphe d'algorithme $\mathcal{A}lg$ est caractérisée par trois paramètres :
- sa durée d'exécution au pire cas (WCET[8]),

[7]Parfois dans certaines applications une opération peut avoir plusieurs exécutions dans un seul cycle d'exécution.
[8]WCET = Worst Case Execution Time.

- sa date de début d'exécution,
- sa date de fin d'exécution.

4.4.1.1 Non-préemption de l'exécution des opérations

Le mode d'exécution cyclique de l'algorithme sur l'architecture définit pour chaque opérateur de l'architecture un ordonnancement statique des opérations qui sont lui allouées. Ceci implique une exécution non-préemptible des opérations.

4.4.1.2 Durées d'exécution des opérations

Étant donné que l'architecture distribuée que nous considérons est hétérogène, c'est-à-dire que ses composants possèdent des caractéristiques différentes, la durée d'exécution d'une même opération peut être différente d'un opérateur de calcul à un autre. Les durées d'exécution de chaque opération sur les opérateurs de calcul de chaque processeur sont contenues dans un tableau $\mathcal{E}xe$ de taille $(n{\times}m)$:

$$\mathcal{E}xe(o_i, p_j) = \{c_{ij}; i \in [1..n], j \in [1..m]\}$$

où c_{ij} est la durée d'exécution de l'opération o_i de O sur l'opérateur p_j de P.

Cette durée d'exécution c_{ij} est une mesure d'exécution au pire cas (WCET) qui dépend des caractéristiques de chaque opérateur, telles que la vitesse du CPU, la présence ou non de mémoires caches, de pipelines… L'estimation de cette durée de pire cas peut être obtenue soit par analyse du code informatique de chaque opération, soit en utilisant des méthodes de mesure dédiées [13, 18].

Par exemple, le tableau 4.1 donne les durées d'exécution $\mathcal{E}xe$ des opérations du graphe d'algorithme $\mathcal{A}lg$ de la figure 4.3 sur les opérateurs de calcul du graphe d'architecture $\mathcal{A}rc$ de la figure 4.8.

		opération								
	$\mathcal{E}xe$	In_1	In_2	A	B	C	D	$init_A$	M_{AB}	Out_1
opérateur	OP_1	1.0	1.0	1.5	1.0	3.0	1.5	0.0	0.0	1.8
	OP_2	1.5	1.5	2.0	1.5	3.5	2.0	0.0	0.0	∞
	OP_3	∞	1.0	1.5	1.0	3.0	1.5	0.0	0.0	1.8

TAB. 4.1 – Durées d'exécution $\mathcal{E}xe$ pour les opérations.

4.4.1.3 Durées des communications entre les opérations

À chaque média (lien ou bus) de communication M_k est associée une durée de communication ou de transfert de données pour chaque dépendance de données $o_i \triangleright o_j$. Cette durée dépend de la quantité de données échangée entre deux opérations et des caractéristiques physiques du médium de communication. Étant donné que l'architecture est hétérogène, cette durée peut être différente d'un médium à un autre. Les durées de transfert de données entre chaque paire d'opérations dépendantes sur chaque médium de communication sont contenues dans un tableau $\mathcal{E}xe$ de taille $(n{\times}m)$:

$$\mathcal{E}xe(o_i \rhd o_j, M_k) = \{c_{k,(ij)}; i \in [1..n], j \in [1..m], k \in [1..q]\}$$

où $c_{k,(ij)}$ est la durée de transfert de données ($o_i \rhd o_j$) sur le média de communication M_k.

Par exemple, le tableau 4.2 donne les durées de transfert des dépendances de données du graphe d'algorithme $\mathcal{A}lg$ de la figure 4.3 sur les média de communication du graphe d'architecture $\mathcal{A}rc$ de la figure 4.8.

		dépendance de donnée					
	$\mathcal{E}xe$	$In_1 \rhd A$	$In_1 \rhd B$	$In_2 \rhd B$	$In_2 \rhd C$	$A \rhd D$	$B \rhd D$
lien	L_{12}	1.75	1.50	1.00	1.00	1.75	1.00
	L_{13}	1.25	1.00	0.50	0.50	1.25	0.50

	$\mathcal{E}xe$	$C \rhd Out_1$	$D \rhd Out_1$	$A \rhd M_{AB}$	$M_{AB} \rhd B$	$init_A \rhd M_{AB}$
lien	L_{12}	1.75	1.50	2.00	2.00	2.00
	L_{13}	1.25	1.00	1.50	1.50	1.50

TAB. 4.2 – Durées de transfert $\mathcal{E}xe$ pour les dépendances de données.

La durée de transfert $c_{k,(ij)}$ est une mesure de transfert au pire cas (WCTT[9]) qui dépend des caractéristiques de chaque médium de communication, telles que la taille de la mémoire SAM et la vitesse de chaque communicateur.

4.4.2 Contraintes liées à l'architecture et au temps réel

Puisque l'architecture que nous visons est distribuée et embarquée, deux contraintes sont à prendre en compte lors de la distribution/ordonnancement d'un algorithme $\mathcal{A}lg$ sur une architecture $\mathcal{A}rc$. Ce sont les contraintes de distribution et les contraintes d'embarquabilité ($\mathcal{D}is$). Elles sont liées à plusieurs critères, souvent d'optimisation :

- *délocalisation de certaines fonctionnalités de l'algorithme sur l'architecture* : dans un système distribué embarqué, afin de limiter le câblage et pour des raisons de fiabilité des données, souvent certains processeurs sont physiquement placés près des capteurs et des actionneurs. Dans ce cas, il est intéressant que les opérations d'entrées/sorties, dédiées à ces capteurs/actionneurs, soient implantées sur ces processeurs ;
- *Processeurs spécialisés* : Certaines opérations ne peuvent être placées que sur des processeurs spécialisés disposant de ressources logicielles ou matérielles particulières. Par exemple des processeurs dédiés aux opérations de traitement du signal.

[9]WCTT = Worst Case Transmision Time.

La spécification des contraintes de distribution et d'embarquabilité consiste à assigner la valeur ∞ aux durées d'exécution de certaines opérations sur certains opérateurs de calcul. Ainsi, $\mathcal{E}xe(o_i, p_j) = \infty$ signifie que l'opération o_i ne peut pas être placée sur l'opérateur p_j.

Enfin, dans un système réactif critique, la réaction à chaque événement d'entrée doit être bornée, c'est-à-dire que le temps de réponse à cet événement ne doit jamais dépasser une certaine valeur critique, appelée *contrainte temps réel* ($\mathcal{R}tc$). Dans notre modèle, une seule contrainte temps réel $\mathcal{R}tc$ est prise en compte, qui est la *latence*, c'est-à-dire que la longueur de la distribution et l'ordonnancement du graphe d'algorithme $\mathcal{A}lg$ sur le graphe d'architecture $\mathcal{A}rc$ doit être inférieure à la borne imposé par la borne $\mathcal{R}tc$.

*« La connaissance est le seul instrument de production
qui n'est pas sujet à la dépréciation. »*
John H. CLARCK

Chapitre 5

État de l'art

Résumé

Les deux disciplines de la tolérance aux fautes et de la fiabilité ont donné lieu à de nombreux développements pour les algorithmes de distribution/ordonnancement temps réel à objectif de tolérance aux fautes et/ou de fiabilité. Ce chapitre présente un état de l'art sur les travaux destinés au développement de ces algorithmes, tout en se limitant aux solutions logicielles pour tolérer des fautes matérielles des processeurs et des média de communications.

5.1 Introduction

Concevoir un système réactif garantissant un fonctionnement sans faute matérielle est impossible, ce qui explique le nombre important de méthodes de tolérance aux fautes matérielles existantes dans la littérature. Puisque ces méthodes diffèrent sur plusieurs critères, tels que l'origine des fautes matérielles (processeurs, média de communication, capteurs, actionneurs, ...), et le type de fautes pris en compte (fautes transitoires, fautes permanentes, ...), nous avons choisi de ne citer dans cette section que deux catégories de ces méthodes : les méthodes de tolérance aux fautes des processeurs et les méthodes de tolérance aux fautes des média de communication. En plus, puisque nous visons des solutions logicielles pour résoudre le problème 2 de la tolérance aux fautes (cf. section 3.2.3, page 39), nous ne présentons que des méthodes basées sur la théorie de l'ordonnancement. Plus particulièrement, nous présentons dans la section 5.2 quelques algorithmes de distribution/ordonnancement temps réel et tolérant aux fautes, et pour chaque algorithme nous donnons son modèle de fautes.

Plusieurs approches basées sur la théorie d'ordonnancement ont été proposées dans la littérature pour concevoir des systèmes fiables. Dans la section 5.3, nous présentons quelques algorithmes de distribution/ordonnancement temps réel et fiables qui permettent de résoudre le problème 3 de la fiabilité, que nous avons présenté dans la section 3.3.3 (page 43).

5.2 Algorithmes de distribution et d'ordonnancement temps réel et tolérants aux fautes

La plupart des algorithmes de distribution/ordonnancement temps réel, développés dans la littérature pour tolérer des fautes matérielles des processeurs et des média de communication, sont basés sur la technique de la *redondance logicielle* [36, 37]. La redondance logicielle est la technique de tolérance aux fautes matérielles la plus adaptée aux systèmes exigeant des contraintes d'embarquabilité fortes.

Trois grandes classes d'algorithme de distribution/ordonnancement temps réel et tolérants aux fautes, basées sur la redondance logicielle, ont été proposées dans la littérature. Ce sont la classe des algorithmes basés sur la *redondance active*, la classe des algorithmes basés sur la *redondance passive*, et la classe des algorithmes basés sur la *redondance hybride*. Nous présentons dans ce qui suit le principe de ces trois classes et dans chaque classe nous donnons quelques exemples d'algorithmes.

5.2.1 Algorithmes basés sur la redondance active

5.2.1.1 Principes

La redondance active peut être utilisée de manière efficace pour tolérer la faute d'un ou plusieurs composants matériels. Elle est bien adaptée aux systèmes critiques et à toutes les hypothèses de défaillance, tels que silence sur défaillances, arrêt sur défaillances, défaillances temporelles et défaillances byzantines.

- Tolérance aux fautes des processeurs

Lorsqu'une faute d'un processeur se produit, tous les composants logiciels implantés sur ce processeur deviennent inactifs, ce qui conduit à une défaillance du système. La redondance active consiste à répliquer *activement* chaque composant logiciel c_i d'un algorithme sur n processeurs distincts pour tolérer au plus k fautes de processeurs.

Remarque 8 *Le nombre n des répliques de chaque composant dépend directement de k et aussi de type des fautes à tolérer. Par exemple, pour tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué sur $n = k + 1$ processeurs distincts.*

Chaque réplique c_i^l de c_i doit recevoir ses données d'entrées de son prédécesseur c_j, soit en un seul exemplaire via une communication intra-processeur, soit en n exemplaires via n communications inter-processeurs. Si l'architecture matérielle est non complètement connectée, les n communications inter-processeurs doivent être implantées sur des routes disjointes.

Définition 37 (Route de communication) *Une route de communication, notée R, est composée d'un ou plusieurs médium de communication : $R = l_{12} \bullet l_{23} \bullet \ldots \bullet l_{ij} \bullet \ldots$; l_{ij} désigne le medium de communication reliant les deux processeurs P_i et P_j.*

Définition 38 (Routes disjointes) *Deux routes R_i et R_j sont dites disjointes ssi elles n'ont pas de média de communication communs :* $(\nexists\, l_{st} \in \mathcal{A}rc \mid l_{st} \in R_i \wedge l_{st} \in R_j)$

Exemple *Dans la figure 5.1b, le composant logiciel A (resp. B) est répliqué en deux copies, qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_2 et P_4) pour tolérer une faute permanente d'un seul processeur. Ici, la réplique B_1 de B reçoit ses données d'entrées en deux exemplaires (message m) via deux routes disjointes : $R_1 = l_{13} \bullet l_{34}$ et $R_2 = l_{24}$. La réplique B_2 de B reçoit ses données d'entrées en un seul exemplaire via une communication intra-processeur.*

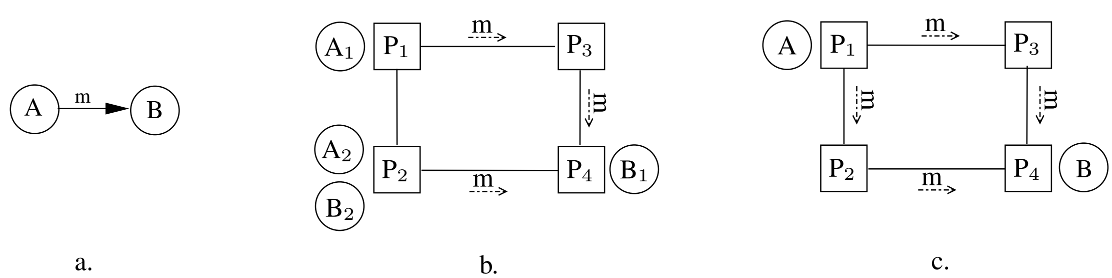


FIG. 5.1 – Exemple de la redondance active.

- Tolérance aux fautes des média de communication

La perte d'un message dans le réseau de communication, due aux fautes des média de communication, peut être tolérée par la transmission de ce même message via plusieurs routes disjointes.

Exemple *Dans la figure 5.1c, le message m est transmis via deux routes disjointes, $R_1 = l_{13} \bullet l_{34}$ et $R_2 = l_{12} \bullet l_{24}$, reliant le processeur P_1 implantant A au processeur P_4 implantant B.*

5.2.1.2 Présentation de quelques algorithmes

Dima et al. ont proposé dans [21] une heuristique basée sur la redondance active pour tolérer certaines configurations de fautes. Ils supposent que les fautes sont des fautes permanentes de processeurs et de média de communication. L'heuristique génère dans un premier temps une distribution/ordonnancement des composants logiciels d'un algorithme sur les composants matériels d'une architecture. Dans un deuxième temps, pour chaque configuration de fautes, elle génère tout d'abord une architecture réduite pour cette configuration, et ensuite, elle génère une distribution/ordonnancement des composants logiciels du même algorithme sur les composants matériels de l'architecture réduite. Une architecture réduite est le résultat de l'exclusion des composants de la configuration de fautes dans l'architecture globale. Ensuite, l'heuristique génère une distribution/ordonnancement globale qui est la combinaison de toutes les distributions/ordonnancements obtenue précédemment.

Un nouveau mécanisme de redondance active de communications est proposé par Banerjea dans [10] et par Dulman et al. dans [23], basé sur le codage de messages en FEC (Forward Error Correction) et des routes disjointes, afin de tolérer plusieurs fautes transitoires de processeurs et de média de communication. Le codage FEC consiste à transmettre des informations redondantes pour chaque message. La première phase de la méthode consiste à coder chaque message m par FEC en plusieurs paquets avec redondance. Ensuite, dans la deuxième phase, chaque paquet d'un message est envoyé vers sa destination via des routes disjointes. Si une partie des paquets d'un message est perdue à cause d'une défaillance le message peut être reconstruit grâce aux informations redondantes envoyées via les autres routes disjointes.

Ramanathan et Shin [67] ont proposé une technique, basée sur la redondance active des communications pour minimiser le coût induit par la retransmission des messages en présence de défaillances. Ils supposent que les fautes sont des fautes transitoires des processeurs et des média de communication. Une date d'échéance et un niveau de criticité sont attribués à chaque message. Pour réduire le coût de la retransmission des messages chaque message est envoyé en parallèle via au moins deux routes disjointes. Le nombre des routes disjointes est défini par la criticité du message et le nombre de processeurs et de média de communication composant chaque route.

Pour réduire le temps de la transmission/retransmission des messages, Kao et al. présentent dans [44] une méthode basée sur la redondance active des messages. La méthode proposée est adaptée aux messages courts. Contrairement à la méthode proposée par Ramanathan et Shin, où le nombre de routes disjointes est différent d'un message à un autre, dans la méthode de Kao et al., les copies de chaque message sont envoyées via un nombre fixe de routes disjointes. La même méthode est utilisée par Kandasamy et al. dans [43] pour tolérer des fautes de processeur et de bus de communication dans une architecture multi-bus. Dans [26], Fragopoulou et Akl utilisent aussi le même principe des routes disjointes pour tolérer les fautes de plusieurs processeurs et de média de communication dans un réseau en étoile. [26, 43, 44] supposent le même modèle de fautes que [67].

Enfin, les solutions proposées dans [10, 23, 26, 43, 44, 67] ne tolèrent que des fautes de processeurs et de média de communication composant une route de communication, et la solution proposée dans [21] ne tolère que certaines configurations de fautes. La solution que nous proposerons dans ce travail est plus générale, car elle peut tolérer *n'importe quelle combinaison arbitraire de plusieurs fautes transitoires* de processeurs et de média de communication.

5.2.2 Algorithmes basés sur la redondance passive

5.2.2.1 Principes

La redondance passive est basée aussi sur la réplication des composants logiciels de l'algorithme en plusieurs copies. À la différence de la redondance active, une seule copie, appelée *copie primaire*, de chaque composant est exécutée, et les autres copies, appelées *copies de sauvegardes*, ne seront exécutées que si une faute provoque une erreur, puis une défaillance du composant matériel implantant la copie primaire. Cela nécessite un mécanisme spécial de détection d'erreurs.

Remarque 9 *Le nombre n de répliques de chaque composant dépend directement de nombre de fautes k et aussi de type de ces fautes. Par exemple, pour tolérer au plus k fautes permanentes de processeurs, chaque composant logiciel est répliqué en une copie primaire et en k copies de sauvegardes, d'où $n = k + 1$.*

- Tolérance aux fautes des processeurs

Afin de tolérer k fautes de processeurs, chaque composant logiciel c_i est répliquée sur n processeurs distincts (voir Remarque 9). Si le processeur implantant la copie primaire défaille, une copie de sauvegarde sera sélectionnée pour remplacer la copie primaire.

Exemple *Dans la figure 5.2a, le composant logiciel A (resp. B) est répliqué en deux copies, qui sont implantées sur deux processeurs distincts P_1 et P_2 (resp. P_4 et P_2) afin de tolérer une faute permanente d'un seul processeur. La copie primaire A_1 envoie le message m à la copie primaire B_1 via la route $R = l_{13} \bullet l_{34}$. Si P_1 défaille, alors P_2 détecte la défaillance de P_1, exécute la copie de sauvegarde A_2 et envoie le message m à P_4 via la seule nouvelle route $R' = l_{24}$, comme cela est montré sur la figure 5.2b.*

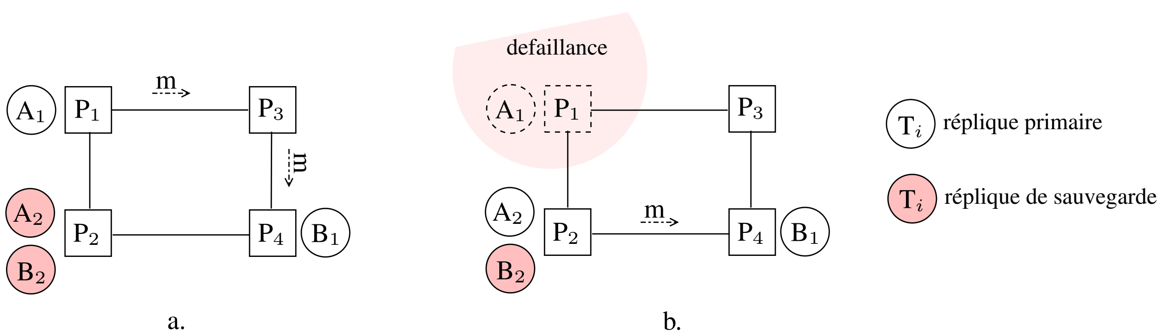


FIG. 5.2 – Tolérance aux fautes des processeurs.

- Tolérance aux fautes des média de communication

La perte d'un message dans le cas de la redondance passive des communications, due aux fautes des média de communication, peut être tolérée par la retransmission de ce message.

Exemple *Dans la figure 5.3a, si le médium l_{13} défaille, alors le processeur P_1 détecte la défaillance de ce médium et envoie le message m à P_4 via la nouvelle route $R' = l_{12} \bullet l_{24}$, comme cela est montré sur la figure 5.3b.*

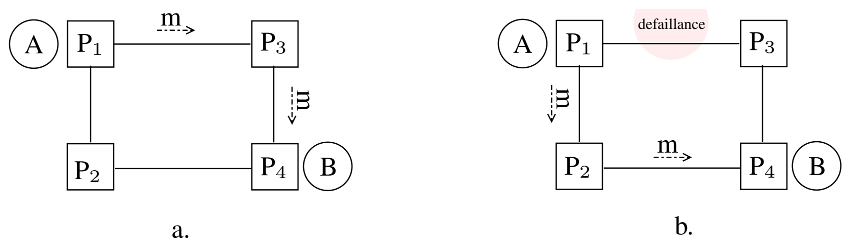


FIG. 5.3 – Tolérance aux fautes des média de communication.

5.2.2.2 Présentation de quelques algorithmes

Oh et Son ont présenté dans [61] une heuristique de distribution/ordonnancement tolérante aux fautes basée sur la redondance passive des composants logiciels d'un algorithme. Ils supposent que les composants logiciels sont indépendants (absence de dépendances de données) et que l'architecture est complètement connectée. Leur modèle de fautes suppose que les processeurs sont de type silence sur défaillances, et que la faute d'un processeur peut être détectée par les autres processeurs. Afin de tolérer une seule faute d'un processeur, ils répliquent chaque composant logiciel en deux copies identiques, une copie primaire et une copie de sauvegarde. Ces deux copies sont allouées temporellement de façon séquentielle sur deux processeurs distincts, et seule la copie primaire est exécutée. En cas de défaillance d'un processeur, la copie de sauvegarde de chaque copie primaire implantée sur ce processeur est exécutée pour tolérer cette défaillance.

Sur le même principe, Xiao et al. ont proposé dans [64] une heuristique de distribution/ordonnancement tolérante aux fautes permanentes des processeurs. À la différence de la méthode de Oh et Son, la solution proposée suppose que les composants logiciels de l'algorithme sont dépendants (présence de dépendances de données), et que plusieurs processeurs peuvent défaillir. La solution que nous proposons est plus générale que ces deux solutions [61] et [64] puisque nous utilisons une *architecture non complètement connectée*, des *composants logiciels dépendants*, et nous tolérons *plusieurs fautes transitoires de processeurs et de média de communication*.

Pour tolérer une faute transitoire d'un médium de communication Zheng et Shin ont proposé dans [85] une méthode basée sur la redondance passive des messages, où chaque message est répliqué en deux copies : copie primaire et copie de sauvegarde. Dans cette méthode, seule la copie primaire de chaque message est envoyée, tandis que la copie de sauvegarde est en attente au cas où la copie primaire serait défaillante. Le même principe est utilisé par Han et Shin dans [38] sauf que cette méthode tolère plusieurs fautes transitoires de processeurs et de média de communication. Comme les routes primaires et les routes de sauvegarde pour chaque message sont pré-calculées, la méthode proposée utilise le multiplexage des routes de sauvegarde de plusieurs messages afin de réduire le surcoût en communication en cas de défaillances. Enfin, ces deux solutions [38, 85] ne tolèrent que les fautes des média de communication et des processeurs utilisés pour le routage dans une route de communication, mais pas les fautes des processeurs émetteurs et récepteurs des communications.

5.2.3 Algorithmes basés sur la redondance hybride

5.2.3.1 Principes

La redondance hybride [25] est une combinaison de la redondance active et passive des composants de l'algorithme. Par exemple, pour tolérer une fautes permanente d'un processeur ou d'un médium de communication, on utilise la redondance active pour les composants logiciels de l'algorithme et la redondance passive pour les communications, comme cela est montré sur la figure 5.4a. Dans cet exemple, les deux composants logiciels A et B sont répliqués activement en deux copies, tandis que la communication (A $\triangleright$ B) ne sera envoyée que par la première réplique A_1 de A (message m) via la route $R = l_{13} \bullet l_{34}$.

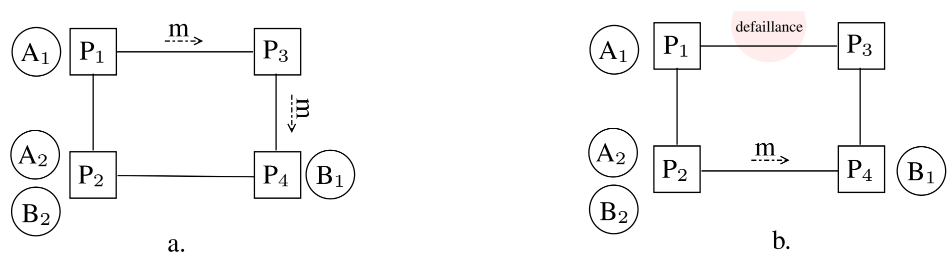


FIG. 5.4 – La redondance hybride.

Dans cette figure, si le médium l_{13} défaille, alors le processeur P_2 détecte la défaillance de ce médium et envoie le message m à P_4 via la nouvelle route $R' = l_{24}$, comme cela est montré sur la figure 5.4b.

5.2.3.2 Présentation de quelques algorithmes

Hashimoto et al. ont proposé dans [39] une heuristique de distribution/ordonnancement basée sur la réplication active des composants logiciels et passive des communications. Ils supposent que l'architecture est complètement connectée et qu'au plus une faute permanente d'un processeur peut affecter le système. L'heuristique réplique activement chaque composant logiciel d'un algorithme sur deux processeurs distincts.

Chevochot et Puaut ont présenté dans [17] une nouvelle approche de tolérance aux fautes des processeurs. Leur modèle de faute suppose que les processeurs sont de type silence sur défaillance, et que les fautes peuvent être permanentes ou transitoires. Ils proposent un outil, appelé Hydra, qui implante un algorithme de transformation de graphe et une heuristique de distribution/ordonnancement tolérante aux fautes. L'algorithme de transformation de graphe transforme chaque composant logiciel sans redondance en un nouveau composant logiciel avec redondance active, passive, et/ou hybride. Ensuite, l'heuristique de distribution/ordonnancement génère une allocation spatiale et temporelle des nouveaux composants logiciels sur les processeurs de l'architecture.

Chen et al. ont proposé dans [15] une méthode hybride basée sur la redondance passive et active des communications. Une date d'échéance et un niveau de tolérance sont attribués à chaque message. Dans cette méthode, chaque message est répliqué activement ou passivement suivant la date d'échéance et le niveau de tolérance du message. Cette méthode ne tolère que des fautes transitoire des routes de communication, la faute d'une route étant causée par la faute d'un de ses liens ou d'un de ses processeurs de routage.

5.2.4 Comparaison

Le tableau 5.1 donne une indication comparative des différentes approches de tolérance aux fautes basées sur la redondance active, passive ou hybride des composants logiciels d'un algorithme.

	Approche		
Critère de comparaison	**Redondance active**	**Redondance passive**	**Redondance hybride**
Surcoût	un surcoût élevé	un surcoût moins élevé	le surcoût dépend du niveau de la réplication active par rapport à la réplication passive
Détection de défaillance	pas besoin de détecter les défaillances	mécanisme spécial de détection de défaillances	mécanisme spécial, coûteux et souvent compliqué
Traitement de défaillances (Temps de réponse)	un temps de réponse prévisible, et généralement rapide dans des architectures offrant un taux élevé de parallélisme	meilleur temps de réponse en absence de défaillances. La défaillance de la réplique primaire peut de manière significative augmenter le temps de réponse	le temps de réponse dépend du niveau de la réplication active par rapport à la réplication passive
Reprise après défaillance	immédiate	non immédiate	non immédiate

TAB. 5.1 – Comparaison entre les trois approches de redondance.

Une propriété intéressante de la redondance active se situe dans le fait qu'une faute n'augmente pas la latence du système temps réel, ce qui n'est pas le cas dans la redondance passive, où la faute de la réplique primaire peut de manière significative augmenter la latence du système. Cependant, la redondance passive présente l'avantage de réduire la surcharge sur les processeurs et sur le réseau de communication, ce qui permet une meilleure exploitation des ressources matérielles offertes par l'architecture.

Le choix d'une telle stratégie de réplication se fait en fonction des contraintes et des besoins applicatifs. Par exemple, les concepteurs de systèmes réactifs tolérants aux fautes préfèrent utiliser la réplication active en cas de défaillances fréquentes des composants matériels, et la réplication passive en cas de nombre élevé de communications.

5.3 Algorithmes de distribution et d'ordonnancement temps réel et fiables

Évaluer la fiabilité d'un système est l'une des préoccupations principales des concepteurs de systèmes sûrs de fonctionnement. Dans ce but, plusieurs approches ont été proposées dans la littérature. Elles diffèrent sur plusieurs critères, tels que le modèle de fautes pris en compte et la méthode de calcul utilisée pour évaluer la fiabilité. Dans ce travail, nous ne nous intéressons qu'aux travaux basés sur la théorie de l'ordonnancement. Suivant le nombre d'objectifs visés par leurs algorithmes de distribution/ordonnancement on peut classer ces approches en deux classes : *algorithmes uni-objectif* et *algorithmes multi-objectifs*.

5.3.1 Algorithmes uni-objectif

Les approches uni-objectif [16, 40, 45, 46, 54, 55, 65, 76] s'intéressent exclusivement au problème de la fiabilité des systèmes. Leur algorithme de distribution/ordonnancement ne vise qu'à maximiser la fiabilité de l'allocation des composants logiciels de l'algorithme sur les composants matériels de l'architecture. Donc, le seul objectif pris en compte est l'objectif de fiabilité.

Dogan et al. ont proposé dans [22] deux algorithmes optimal et sub-optimal pour le problème de distribution/ordonnancement temps réel. Leur modèle suppose que les fautes sont des fautes permanentes des processeurs et des média de communication. Ils supposent aussi que la défaillance des composants matériels (processeurs et média de communication) suit une loi exponentielle à taux de défaillance constant. L'objectif de ces algorithmes est de générer une allocation fiable des composants logiciels de l'algorithme sur les composants matériels de l'architecture, c'est-à-dire, générer une allocation qui maximise leur fiabilité. La mesure de fiabilité calculée par cette heuristique représente la probabilité que chaque composant logiciel fonctionne correctement durant son exécution.

He et al. ont proposé dans [40] deux heuristiques de distribution/ordonnancement temps réel pour systèmes hétérogènes, appelées MCMS (Minimum Cost Match Schedule) et PRMS (Progressive Reliability Maximization Schedule). Ils utilisent le même modèle de fautes que [22]. Le but de ces heuristiques est de générer l'allocation la plus fiable possible des composants logiciels de l'algorithme sur les composants matériels de l'architecture distribuée, tout en respectant des contraintes temps réel.

5.3.2 Algorithmes multi-objectifs

À la différence des approches uni-objectif, les approches multi-objectifs [6, 64, 66, 73] s'intéressent au problème de la fiabilité et aussi au problème de la tolérance aux fautes. Donc, ces approches visent deux objectifs qui sont la maximisation de la fiabilité et la tolérance aux fautes matérielles.

Shatz et al. ont proposé dans [73] quatre algorithmes hors-ligne de distribution/ordonnancement dans le but de générer des allocations fiables et tolérantes aux fautes. Leur modèle suppose que les fautes sont des fautes permanentes des processeurs et des média de communication. Ils supposent que la défaillance des composants matériels (processeurs et média de communication) suit une loi exponentielle à taux de défaillance constant et que chaque composant matériel défaillant est remplacé immédiatement après sa défaillance. Contrairement à la méthodologie que nous présenterons au chapitre 8, ces algorithmes ne considèrent que l'objectif de la maximisation de la fiabilité dans leur fonction de coût pour trier les composants logiciels à ordonnancer. La mesure de fiabilité calculée par ces algorithmes représente la probabilité que le système fonctionne correctement durant la *totalité* de la mission. En effet, le coût d'exécution d'un composant logiciel est la somme des durées de toutes ses exécutions durant cette mission.

Xiao et al. ont proposé dans [64] une heuristique de distribution/ordonnancement temps réel à deux objectifs. Le premier objectif consiste à générer une allocation tolérante à une seule faute permanente d'un processeur, tandis que le deuxième objectif consiste à maximiser la fiabilité de cette allocation. Leur modèle suppose que la défaillance des processeurs suit une loi exponentielle à taux de défaillance constant. La tolérance aux fautes des processeurs est obtenue par l'utilisation de la redondance passive des composants logiciels. La méthodologie que nous présenterons au chapitre 8 utilise quant à elle la redondance active des composants logiciels. L'heuristique que proposent Xiao et al. est une heuristique glouton de type ordonnancement de liste [83], basée sur une fonction de coût à deux objectifs (temps réel et fiabilité) pour trier les composants logiciels à ordonnancer. Au contraire de [73], la mesure de fiabilité calculée par cette heuristique représente la probabilité que chaque composant logiciel fonctionne correctement durant uniquement son exécution, ce qui ne nécessite donc pas d'estimer la durée totale de la mission du système.

Auluck et Agrawal ont présenté dans [6] une heuristique de distribution/ordonnancement, appelée RDRTPSA (Reliability Driven Real Time Periodic Scheduling Algorithms). Ils utilisent le même modèle de fautes et la même fonction d'évaluation de la fiabilité que [73]. Cependant, à la différence de [73], Auluck et Agrawal supposent que les média de communication sont sans fautes, et la mesure de fiabilité calculée par leur fonction de d'évaluation représente la probabilité que le système fonctionne correctement durant un cycle d'exécution du système. La taille d'un cycle d'exécution représente la longueur de l'allocation générée par leur heuristique. Enfin, la tolérance aux fautes des processeurs est obtenue par l'utilisation de la redondance passive des composants logiciels.

5.3.3 Discussion

L'algorithme de distribution/ordonnancement que nous proposons dans le chapitre 8 est classé parmi les algorithmes uni-objectif. Il peut être facilement étendu pour tolérer en plus des fautes matérielles, ce qui en ferait un algorithme multi-objectifs. Ceci est du à l'utilisation de la redondance active dans notre solution.

5.4 Conclusion

Nous avons présenté dans ce chapitre différents algorithmes de distribution/ordonnancement, existants dans la littérature, pour générer des distributions/ordonnancements tolérants aux fautes et fiables. Les algorithmes que nous avons présentés sont tous basés sur la redondance logicielle des composants logiciels. Nous présenterons dans ce qui suit, trois méthodologies pour la génération de ce type de distribution/ordonnancement. Elles sont basées sur la redondance active et hybride des composants logiciels.

« La difficulté avec la tolérance, vient de ce qu'elle parait tout à la fois nécessaire et impossible. »
Bernard WILLIAMS

Chapitre 6

Méthodologie AAA-TP pour des architectures à liaisons point-à-point

Résumé

Ce chapitre présente une nouvelle méthodologie pour la génération automatique de distribution/ordonnancement tolérant aux fautes pour systèmes distribués réactifs embarqués. La méthodologie proposée est adaptée aux architectures matérielles munies d'un réseau de communication composé uniquement de liaisons point-à-point. Elle permet de tolérer une ou plusieurs fautes temporelles des processeurs et des liens de communication. La tolérance aux fautes est obtenue hors-ligne en deux phases. La première phase s'appuie sur un formalisme de graphes pour transformer une spécification d'un graphe d'algorithme sans redondances en une spécification avec redondances et relations d'exclusion. La deuxième phase consiste à allouer spatialement et temporellement les composants logiciels de ce nouveau graphe d'algorithme sur les composants matériels d'un graphe d'architecture.

6.1 Présentation du problème bi-objectifs de tolérance aux fautes et de prédictibilité

Rappelons que notre travail s'inscrit dans l'objectif global de la conception de systèmes distribués réactifs embarqués à contraintes strictes. Notre problématique vise dans ce chapitre deux caractéristiques particulières de ces systèmes qui sont la tolérance aux fautes et la prédictibilité. Plus particulièrement, nous allons traiter le problème de la génération automatique de distributions/ordonnancements temps réel prédictibles et tolérantes aux fautes. La tolérance aux fautes consiste à introduire dans la distribution/ordonnancement un ensemble de redondances pour que le système continue à fonctionner en présence de certaines défaillances matérielles. La prédictibilité consiste à vérifier hors-ligne que les contraintes temporelles sont respectées en absence et en présence de défaillances.

Nous ne nous intéressons, dans ce chapitre, qu'au problème de distribution/ordonnancement lié aux architectures matérielles munies d'un réseau de communication composé uniquement de liaisons point-à-point. Afin de bien présenter le problème de distribution/ordonnancement tolérante aux fautes et prédictible, nous présentons tout d'abord notre modèle de fautes.

6.1.1 Modèle de fautes

Nous ne nous intéressons dans ce travail qu'aux techniques de tolérance aux fautes matérielles basées sur des solutions logicielles. Étant donné que chaque nouvelle solution pour la tolérance aux fautes est liée aux hypothèses de défaillances définies par son modèle de fautes, dans notre modèle de faute nous supposons que :

Hypothèse 2 *Les capteurs sont fiables, c'est-à-dire que les valeurs issues des capteurs (opérations d'entrées) sont supposées correctes.*

Hypothèse 3 *Les actionneurs sont fiables, c'est-à-dire que les actionneurs (opérations de sorties) réagissent aux évènements d'entrées en produisant des actions de sortie adéquates.*

Hypothèse 4 *Les fautes matérielles sont des fautes des opérateurs de calculs et des fautes des liens de communication. La faute d'un lien de communication peut être la faute d'un de ses composants (cf. section 4.3.1, page 56).*

Hypothèse 5 *Le système accepte au plus $\mathcal{N}pf$[1] fautes des opérateurs de calcul et $\mathcal{N}lf$[2] fautes de liens de communications dans un cycle d'exécution de son algorithme sur son architecture.*

Hypothèse 6 *Le réseau de communication physique n'est jamais partitionné, même en présence de $\mathcal{N}pf+\mathcal{N}lf$ fautes actives dans un cycle d'exécution de son algorithme sur son architecture.*

Hypothèse 7 *Les fautes des composants matériels (opérateurs de calcul et liens de communication) sont des fautes transitoires, c'est-à-dire que la durée de l'activation d'une faute d'un composant est limitée dans le temps.*

Hypothèse 8 *Les opérateurs de calculs et les liens de communication sont à défaillances temporelles, c'est-à-dire que les valeurs calculées par les opérateurs de calcul sont soit correctes et délivrées à temps, soit correctes et délivrées trop tôt, trop tard ou infiniment tard.*

Nous supposons aussi que le logiciel est sans fautes (hypothèse 1, section 3.2, page 39). Enfin, notre hypothèse de défaillances temporelles couvre les deux hypothèses de défaillances les plus utilisées, qui sont : hypothèse de défaillances par omission et hypothèse de silence sur défaillances [68].

[1] $\mathcal{N}pf$ = Number of Processor Failures.
[2] $\mathcal{N}lf$ = Number of Link Failures.

6.1.2 Données du problème

Le but de ce chapitre est de résoudre le problème de la recherche d'une distribution/ordonnancement des composants logiciels du graphe d'algorithme sur les composants matériels du graphe d'architecture, qui doit *tolérer des fautes matérielles* des opérateurs de calcul et des liens de communication, tout en *minimisant la longueur* de cette distribution/ordonnancement dans le but de satisfaire la contrainte temps réel $\mathcal{R}tc$ en absence et en présence de défaillances. Ce problème a été abordé d'une façon générale dans le chapitre 3 (problème 2, page 40). Plus particulièrement, ce problème de distribution/ordonnancement tolérante aux fautes/prédictible peut être formalisé comme suit :

Problème 4 *Étant donnés :*

- *une architecture matérielle hétérogène $\mathcal{A}rc$ composée d'un ensemble P d'opérateurs de calcul et d'un ensemble L de liens de communication (cf. section 4.3.1, page 56) :*

$$P = \{\ldots, p_i, \ldots, p_j, \ldots\}, L = \{\ldots, l_{i,j}, \ldots\}$$

- *un algorithme $\mathcal{A}lg$ composé d'un ensemble E de dépendances de données et d'un ensemble O d'opérations (cf. section 4.2, page 48) :*

$$O = \{\ldots, o_i, \ldots, o_j, \ldots\}, E = \{\ldots, (o_i \triangleright o_j), \ldots\}$$

- *des caractéristiques d'exécution $\mathcal{E}xe$ des composants de $\mathcal{A}lg$ sur les composants de $\mathcal{A}rc$ (cf. section 4.4.1, page 58),*
- *un ensemble de contraintes matérielles $\mathcal{D}is$ (cf. section 4.4.2, page 60),*
- *une contrainte temps réel $\mathcal{R}tc$ (cf. section 4.4.2, page 60),*
- *un critère de minimisation de la longueur de la distribution/ordonnancement,*
- *un nombre $\mathcal{N}pf$ de fautes d'opérateurs de calcul et un nombre $\mathcal{N}lf$ de fautes de liens de communication qui peuvent causer la défaillance du système,*

il s'agit de trouver une application $\mathcal{A}$ qui place chaque opération (resp. dépendance de données) de $\mathcal{A}lg$ sur un opérateur (resp. un lien) de $\mathcal{A}rc$, et qui lui assigne un ordre d'exécution t_k sur son opérateur (resp. un lien) :

$$\begin{array}{rlcl} \mathcal{A}: & \mathcal{A}lg & \longrightarrow & \mathcal{A}rc \\ & \mathcal{A}lg_i & \longmapsto & \mathcal{A}(\mathcal{A}lg_i) = (\mathcal{A}rc_j, t_k) \end{array}$$

qui respecte $\mathcal{D}is$, minimise la longueur de la distribution/ordonnancement afin de satisfaire $\mathcal{R}tc$, et tolére $\mathcal{N}pf$+$\mathcal{N}lf$ fautes d'opérateurs et de liens de communication.

Remarque 10 *Dans la suite de ce chapitre nous désignons par le mot « processeur » son opérateur de calcul.*

6.2 Principe général de la méthodologie AAA-TP

Le problème 4 peut être vu comme une combinaison de deux problèmes : un problème de temps réel (sans contrainte de tolérance aux fautes), notée $\mathcal{P}rb^{\mathcal{R}tc}$, et un problème de tolérance aux fautes (sans contrainte temps réel), notée $\mathcal{P}rb^{\mathcal{F}t}$. Le problème $\mathcal{P}rb^{\mathcal{R}tc}$ est un *problème d'optimisation* [11], puisque il s'agit ici de trouver une solution optimale, c'est-à-dire une solution valide qui minimise la longueur de la distribution/ordonnancement. Ce problème d'optimisation a été démontré par Lenstra et al. [53] comme étant NP-difficile :

Théorème 1 (Lenstra et al.) *Soient un algorithme constitué de plusieurs composants logiciels dépendants, et une architecture matérielle hétérogène constituée de plusieurs processeurs. La distribution/ordonnancement d'un tel algorithme sur une telle architecture visant la minimisation de la longueur de la distribution/ordonnancement est un problème NP-difficile.*

Le problème $\mathcal{P}rb^{\mathcal{R}tc}$ ne peut être résolu de façon exacte en complexité polynomiale. De plus, ce qui nous intéresse n'est pas d'avoir une distribution/ordonnancement de longueur minimale mais plutôt qui respecte la contrainte temps réel $\mathcal{R}tc$. C'est pourquoi nous nous sommes attachés dans ce travail à des solutions approchées pour résoudre ce problème en complexité polynomiale. Donc, l'heuristique de distribution/ordonnancement présentée dans la section 2.6 est la solution approchée la plus adaptée à ce problème.

Le problème de tolérance aux fautes $\mathcal{P}rb^{\mathcal{F}t}$ peut être résolu par l'utilisation de plusieurs techniques logicielles [50]. Étant donné que nous visons des systèmes embarqués, les techniques basées sur la redondance logicielle sont les techniques qui nous intéressent le plus dans ce travail.

Principe 1 (Redondance logicielle) *Afin de satisfaire les contraintes physiques et financières exigées par les systèmes embarqués, nous nous intéressons uniquement aux solutions logicielles, basées sur la redondance, c'est-à-dire que nous utilisons au mieux la redondance matérielle existante dans l'architecture du système sans essayer de rajouter des ressources physiques supplémentaires.*

Le problème $\mathcal{P}rb^{\mathcal{F}t}$ est un problème P, qui peut être résolu en temps polynômial, puisque il s'agit ici, dans un premier temps de transformer un algorithme $\mathcal{A}lg$ sans redondance en un nouvel algorithme $\mathcal{A}lg^*$ avec redondances logicielles, puis dans un deuxième temps de distribuer/ordonnancer les composants logiciels de $\mathcal{A}lg^*$ sur une architecture $\mathcal{A}rc$, tout en respectant les contraintes matérielles $\mathcal{D}is$ mais sans chercher à optimiser le résultat. Ces deux étapes peuvent être effectuées en temps polynômial, donc le problème $\mathcal{P}rb^{\mathcal{F}t}$ est bien P.

Si on considère les deux problèmes $\mathcal{P}rb^{\mathcal{R}tc}$ et $\mathcal{P}rb^{\mathcal{F}t}$ ensemble, donc le problème 4, il est évident que ce problème est NP-difficile.

Théorème 2 *Soient un algorithme constitué de plusieurs composants logiciels dépendants, et une architecture matérielle hétérogène constituée de plusieurs processeurs. La distribution/ordonnancement d'un tel algorithme sur une telle architecture visant l'objectif de la minimisation de cette distribution/ordonnancent et l'objectif de la tolérance aux fautes est un problème NP-difficile.*

Afin de résoudre le problème 4 en temps polynômial, nous proposons une méthodologie, appelée AAA-TP[3], qui essaye de trouver une solution valide. Cette solution doit vérifier la contrainte temps réel $\mathcal{R}tc$ avant la mise en exploitation du système et ceci en présence d'au plus $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances matérielles. La méthodologie AAA-TP implante une solution logicielle basée sur la redondance active.

Principe 2 (Redondance active) *La redondance active des composants logiciels a été choisie comme technique de redondance car elle nous paraît la plus appropriée pour pouvoir atteindre hors-ligne les deux objectifs de tolérance aux fautes et de prédictiblité.*

Principe 3 (Masquage des erreurs) *Nous utilisons une stratégie de compensation des erreurs, basée sur la redondance active, qui permet de masquer au plus $\mathcal{N}pf$+$\mathcal{N}lf$ erreurs matérielles.*

Le masquage des erreurs présente l'inconvénient de surcoût en nombre d'exécution des composants logiciels d'un algorithme $\mathcal{A}lg$, mais il apporte deux avantages qui sont la prédictibilité et l'absence d'un mécanisme spécial de détection des erreurs (car nous nous plaçons dans un système sans réparation). Puisque nous visons des architectures non forcément complètement connectées, l'utilisation d'un tel mécanisme de détection des erreurs peut augmenter significativement la latence (le délai entre l'activation d'une faute et sa détection), ce qui peut avoir un effet sur le non respect de la contrainte temps réel $\mathcal{R}tc$ en présence de défaillances. Il nous semble donc que la redondance active est la solution adéquate pour résoudre le problème 4 visant des architectures à liaisons point-à-point.

Principe 4 (Tolérance aux fautes arbitraires) *La méthodologie AAA-TP peut générer une distribution/ordonnancement tolérante à n'importe quelle combinaison d'au plus $\mathcal{N}pf$ fautes de processeurs et d'au plus $\mathcal{N}lf$ fautes de liens de communication.*

La méthodologie AAA-TP peut générer hors-ligne et en deux phases successives une distribution/ordonnancement tolérante aux fautes, comme cela est montré sur la figure 6.1. La première phase, que nous appelons *phase d'initialisation*, consiste à transformer le graphe d'algorithme $\mathcal{A}lg$ en un nouveau graphe d'algorithme $\mathcal{A}lg^*$ avec redondances logicielles et un ensemble de *relations d'exclusion* $\mathcal{E}xcl$. L'ensemble $\mathcal{E}xcl(o_i)$ (resp. $\mathcal{E}xcl(o_i \triangleright o_j)$) est l'ensemble des opérations répliques de l'opération o_i (resp. de la dépendance de données $o_i \triangleright o_j$) qui doivent être placées sur des processeurs distincts (resp. sur des routes disjointes). La deuxième phase, que nous appelons *phase d'adéquation*, est réalisée par une heuristique d'adéquation, qui consiste à mettre en correspondance de manière efficace le nouveau graphe d'algorithme $\mathcal{A}lg^*$ sur le graphe d'architecture $\mathcal{A}rc$ pour réaliser une allocation optimisée.

[3]AAA-TP = Adéquation Algorithme Architecture Tolérante aux fautes pour des architectures à liaisons Point-à-point.

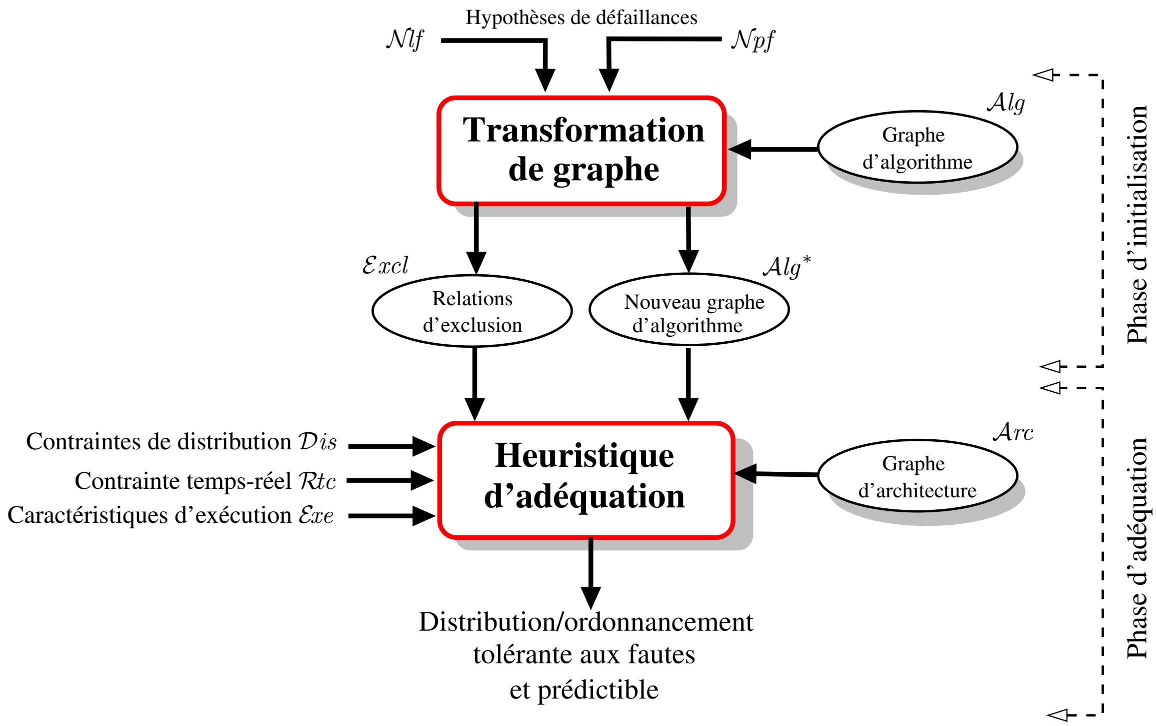


FIG. 6.1 – Méthodologie AAA-TP.

6.3 Phase d'initialisation : transformation de graphe

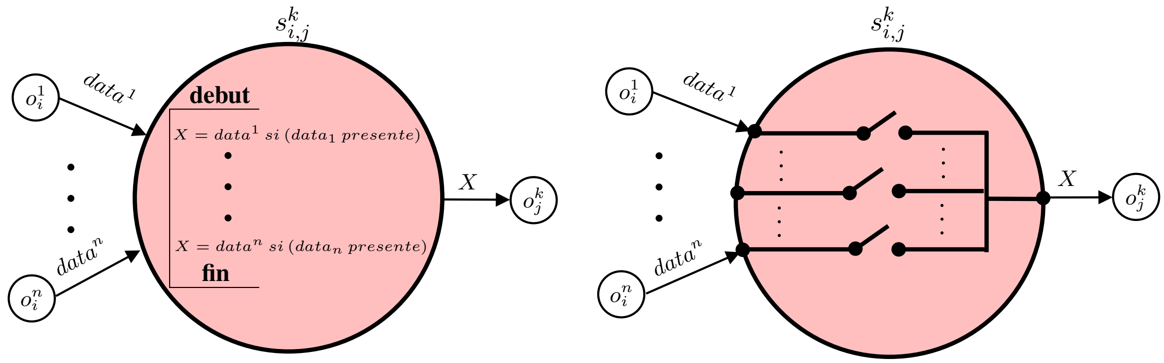
L'objectif de cette phase d'initialisation est de répliquer les composants logiciels du graphe d'algorithme $\mathcal{A}lg$ afin de tolérer $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}lf$ fautes de liens de communication. Les composants logiciels à répliquer sont les opérations de calcul, d'entrée/sortie et de mémoire, et les dépendances de données. Nous transformons donc le graphe flot de données $\mathcal{A}lg$ en un nouveau graphe $\mathcal{A}lg^*$ avec redondances logicielles, et nous lui adjoignons un ensemble de relations d'exclusion $\mathcal{E}xcl$.

6.3.1 Notations et définitions

Le nouveau graphe d'algorithme $\mathcal{A}lg^*$ est aussi un graphe orienté. Un sommet de ce graphe est soit une *opération réplique*, soit une *opération de contrôle*.

Définition 39 (Opération réplique) *Une opération réplique o_i^k de $\mathcal{A}lg^*$ représente la k^{ieme} réplique de l'opération o_i de $\mathcal{A}lg$. Puisque le système est sans faute logicielle (hypothèse 1, section 3.2), toutes les répliques d'une même opération sont identiques, c'est-à-dire qu'elles ont le même code informatique.*

Définition 40 (Opération de contrôle) *Une opération de contrôle, appelée aussi* « switch », *est un composant logiciel caractérisé par ses dépendances de données d'entrées qui proviennent de toutes les opérations répliques d'une même opération de $\mathcal{A}lg$, et par son unique dépendance de données de sortie. Son rôle est de sélectionner sa dépendance de données de sortie parmi toutes ses dépendances de données d'entrées. Concrètement, elle réalise une opération de routage des données avec vérifications. Son code informatique a la forme de la figure 6.2.*



a. Représentation informatique b. Représentation graphique

FIG. 6.2 – Forme d'une opération de contrôle.

Dans l'exemple de la figure 6.2, l'opération de contrôle s^k_{ij} vérifie tout d'abord la présence, sur son processeur, d'au moins une copie $data^i$ des données $data$, de la dépendance $(o_i \triangleright o_j)$, envoyées par les processeurs implantant les répliques de o_i, et ensuite elle envoie une seule copie de ces données au processeur implantant la réplique o^k_j.

Hypothèse 9 *Nous supposons que la durée d'exécution d'une opération de contrôle est nulle sur chaque processeur du graphe d'architecture $\mathcal{A}rc$.*

Les arcs du nouveau graphe $\mathcal{A}lg^*$ sont les dépendances de données de $\mathcal{A}lg$ répliquées en plusieurs copies identiques. Ainsi, chaque dépendance de données $(o^l_i \triangleright s^k_{i,j})$ de $\mathcal{A}lg^*$, réplique de $(o_i \triangleright o_j)$, a les mêmes caractéristiques (taille et type de données). De même, chaque dépendance de données $(s^k_{i,j} \triangleright o^k_j)$, réplique de $(o_i \triangleright o_j)$, a les mêmes caractéristiques.

Remarque 11 *Dans la suite de ce chapitre, nous notons l'ensemble des dépendances de données* $\{(o^1_i \triangleright o^k_j), \ldots, (o^n_i \triangleright o^k_j)\}$, *répliques de* $(o_i \triangleright o_j)$, *par* $(o^*_i \triangleright o^k_j)$.

Les *relations d'exclusion* $\mathcal{E}xcl$, engendrées par cette transformation de graphe, définissent trois types de contraintes de distribution des composants logiciels de $\mathcal{A}lg^*$ sur les composants matériels de $\mathcal{A}rc$, qui sont :

- *Contraintes d'exclusion entre opérations répliques* : toutes les opérations répliques o_i^k d'une même opération o_i de $\mathcal{A}lg$ doivent être exclusives ;

  Définition 41 (Opérations exclusives) *Deux opérations o_i^1 et o_i^2 sont dites exclusives ssi elles sont deux répliques identiques d'une même opération o_i et sont placées sur deux processeurs distincts. On note $\|o_i^1, o_i^2\|$ cette exclusion.*

- *Contraintes d'exclusion entre dépendances de données* : toutes les dépendances de données $(o_i^* \triangleright s_{i,j}^k)$ répliques d'une même dépendance de données $(o_i \triangleright o_j)$ de $\mathcal{A}lg$ doivent être exclusives ;

  Définition 42 (Dépendances exclusives) *Deux dépendances de données $(o_i^l \triangleright o_j^k)$ et $(o_i^m \triangleright o_j^k)$ sont dites exclusives ssi elles sont deux répliques identiques d'une même dépendance $(o_i \triangleright o_j)$ et sont placées sur deux routes disjointes (définition 38, page 65). On note $\|(o_i^l \triangleright o_j^k), (o_i^m \triangleright o_j^k)\|$ cette exclusion.*

- *Contraintes de placement intra-processeur* : l'opération de contrôle $s_{i,j}^k$ et sa seule opération successeur o_j^k doivent être placées sur un même processeur, et donc la dépendance de données $(s_{i,j}^k \triangleright o_j^k)$, réplique de $(o_i \triangleright o_j)$, doit être placée comme une opération de communication intra-processeur.

Un exemple d'une telle transformation est donné par la figure 6.3, où le graphe d'algorithme $\mathcal{A}lg$ de la figure 6.3a est transformé en le nouveau graphe d'algorithme $\mathcal{A}lg^*$ de la figure 6.3b, et ceci afin de tolérer une faute d'un processeur ($\mathcal{N}pf$=1).

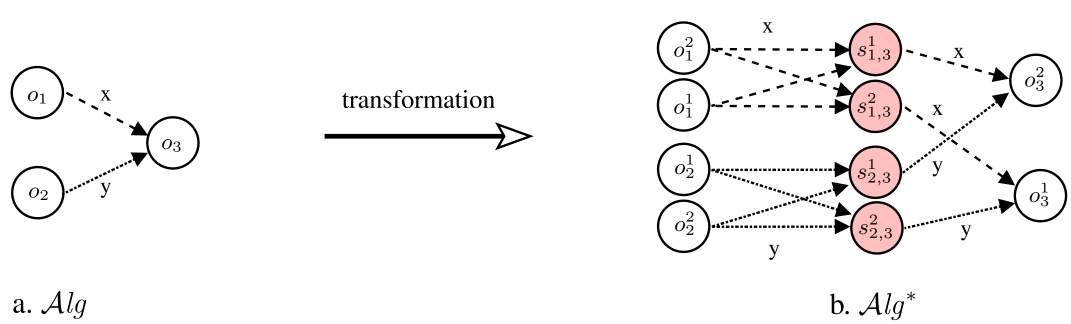


FIG. 6.3 – Exemple de transformation d'un graphe d'algorithme.

Afin de bien expliquer cette phase de transformation de graphe d'algorithme pour tolérer les fautes des processeurs et des liens de communication, nous avons choisi de la présenter tout d'abord pour tolérer uniquement les fautes des processeurs, puis pour tolérer uniquement les fautes des liens de communication, et enfin pour tolérer les deux.

6.3.2 Tolérance aux fautes des processeurs

Nous supposons ici que les communications sont fiables, c'est-à-dire que le système est sans fautes des liens de communication. Donc, le seul objectif est de transformer le graphe d'algorithme $\mathcal{A}lg$ en un nouveau graphe d'algorithme $\mathcal{A}lg^*$ avec redondances logicielles pour tolérer $\mathcal{N}pf$ fautes de processeurs. Cette transformation se fait par AAA-TP en deux temps.

Dans un premier temps, afin de tolérer au plus $\mathcal{N}pf$ fautes des processeurs, il est nécessaire de placer chaque opération de $\mathcal{A}lg$ sur $\mathcal{N}pf$+1 processeurs distincts de $\mathcal{A}rc$. Donc :

- chaque opération o_i de $\mathcal{A}lg$ est répliquée dans $\mathcal{A}lg^*$ en $\mathcal{N}pf$+1 répliques exclusives o_i^1, o_i^2, $\ldots$, $o_i^{\mathcal{N}pf+1}$; l'ensemble de toutes les répliques de o_i est noté $\mathcal{R}ep(o_i)$. Par exemple, dans la figure 6.4b, afin de tolérer une seule faute d'un processeur (i.e., $\mathcal{N}pf$=1), les deux opérations o_1 et o_2, de la figure 6.4a, sont répliquées dans $\mathcal{A}lg^*$ en deux répliques chacune.
- ensuite, puisque les opérations répliques $\mathcal{R}ep(o_i)$ de chaque opération o_i doivent envoyer en parallèle leurs données de sortie ($o_i^* \triangleright o_j^k$) à chaque opération successeur o_j^k, l'opération o_j^k doit tout d'abord *vérifier* la validité temporelle de chaque réplique, et elle doit ensuite *sélectionner* parmi ces répliques la meilleure réplique. Afin de rendre la phase de la tolérance aux fautes transparente aux utilisateurs, ces deux tâches de vérification et de sélection sont réalisées par un nouveau composant logiciel, appelé « opération de contrôle » (définition 40). Par conséquent, entre chaque paire d'ensembles $\mathcal{R}ep(o_i)$ et $\mathcal{R}ep(o_j)$, où $(o_i \triangleright o_j) \in \mathcal{A}lg$, est ajouté dans $\mathcal{A}lg^*$ un ensemble $\mathcal{R}ep(s_{i,j})$ de $\mathcal{N}pf$+1 opérations de contrôle. Par exemple, dans la figure 6.4b, deux opérations de contrôle $s_{1,2}^1$ et $s_{1,2}^2$ sont ajoutées dans le nouveau graphe d'algorithme $\mathcal{A}lg^*$.

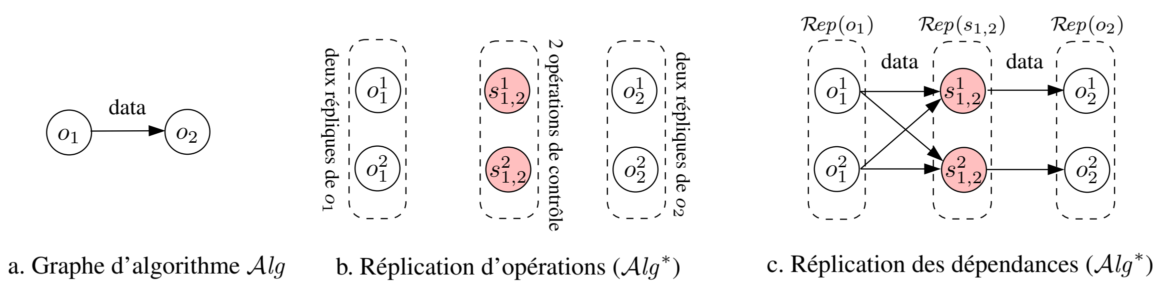


FIG. 6.4 – Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 0$.

Remarque 12 *Les opérations de contrôle peuvent être utilisées aussi comme des voteurs, ceci afin de tolérer, en plus des fautes temporelles, des fautes fonctionnelles (définition 18, page 35).*

Dans un deuxième temps, chaque opération o_i^k réplique de o_i doit envoyer à toutes les répliques $\mathcal{R}ep(o_j)$ de o_j, successeur de o_i, les données ($o_i \triangleright o_j$) par l'intermédiaire des opérations de contrôle $\mathcal{R}ep(s_{i,j})$. Donc :

- la dépendance de données ($o_i \triangleright o_j$) de $\mathcal{A}lg$ est répliquée dans $\mathcal{A}lg^*$ entre chaque opération de $\mathcal{R}ep(o_i)$ et chaque opération $\mathcal{R}ep(s_{i,j})$. Ce qui ajoute à $\mathcal{A}lg^*$ un ensemble de $\mathcal{N}pf$+1 dépendances exclusives de données pour chaque $s^k_{i,j}$. Par exemple, dans la figure 6.4c, la dépendance de donnée ($o_1 \triangleright o_2$), de la figure 6.4a, est répliquée entre chaque opération de $\mathcal{R}ep(o_1)$ et chaque opération de $\mathcal{R}ep(s_{1,2})$. Les dépendances de données de l'ensemble $\{(o^1_1 \triangleright s^1_{1,2}), (o^2_1 \triangleright s^1_{1,2})\}$ (resp. l'ensemble $\{(o^1_1 \triangleright s^2_{1,2}), (o^2_1 \triangleright s^2_{1,2})\}$) sont exclusives.

- ensuite, la dépendance de données ($o_i \triangleright o_j$) est répliquée dans $\mathcal{A}lg^*$ entre chaque k^{ieme} réplique de $s_{i,j}$ et chaque k^{ieme} réplique de o_j. Ces répliques doivent être placées comme des communications intra-processeur. Par exemple, dans la figure 6.4c, la dépendance de données ($o_1 \triangleright o_2$), de la figure 6.4a, est répliquée entre $s^1_{1,2}$ et o^1_2, et entre $s^2_{1,2}$ et o^2_2.

Enfin, cette transformation génère une liste de relations d'exclusion $\mathcal{E}xcl$ entre opérations et aussi entre dépendances de données. Par exemple, pour la figure 6.4c :

$$\mathcal{E}xcl = \{\|o^1_1, o^2_1\|, \|o^1_2, o^2_2\|\} \cup \{\|(o^1_1 \triangleright s^1_{1,2}), (o^2_1 \triangleright s^1_{1,2})\|, \|(o^1_1 \triangleright s^2_{1,2}), (o^2_1 \triangleright s^2_{1,2})\|\}$$

La figure 6.5 représente la transformation du graphe d'algorithme $\mathcal{A}lg$ de la figure 6.4a dans le cas général où $\mathcal{N}pf \geq 1$.

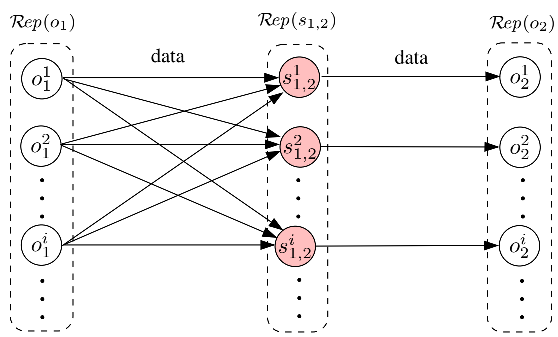


FIG. 6.5 – Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf \geq 1$ et $\mathcal{N}lf = 0$.

6.3.3 Tolérance aux fautes des liens de communication

Nous supposons ici que les processeurs sont fiables, c'est-à-dire que le système est sans faute des processeurs. Le seul objectif est donc de transformer le graphe d'algorithme $\mathcal{A}lg$ en un nouveau graphe d'algorithme $\mathcal{A}lg^*$ avec redondances logicielles pour tolérer $\mathcal{N}lf$ fautes de liens de communication. Cette transformation se fait par AAA-TP en une seule étape.

Afin de tolérer au plus $\mathcal{N}lf$ fautes de liens de communication, il est nécessaire de placer chaque dépendance de données de $\mathcal{A}lg$ sur $\mathcal{N}lf$+1 routes disjointes de $\mathcal{A}rc$. Cependant, il n'est pas besoin de répliquer les opérations de $\mathcal{A}lg$ puisque les processeurs sont fiables, ce qui implique que les opérations de $\mathcal{A}lg^*$ sont les mêmes opérations de $\mathcal{A}lg$, et que chaque dépendance de données ($o_i \triangleright o_j$) de $\mathcal{A}lg$ est répliquée en $\mathcal{N}lf$+1 dépendances exclusives dans $\mathcal{A}lg^*$. Comme dans le cas de la tolérance aux fautes des processeurs, chaque opération doit vérifier la validité temporelle de chaque dépendance réplique, puis elle doit *sélectionner* parmi ces répliques la meilleure réplique. Donc, entre chaque paire d'opérations dépendantes o_i et o_j, est ajoutée dans $\mathcal{A}lg^*$ une opération de contrôle $s_{i,j}$.

La figure 6.6 représente la transformation du graphe d'algorithme $\mathcal{A}lg$ de la figure 6.4a dans le cas où $\mathcal{N}lf \geq 1$. Cette transformation génère une liste de relations d'exclusion $\mathcal{E}xcl$ entre les

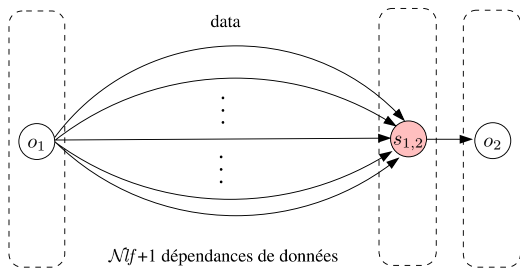


FIG. 6.6 – Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf = 0$ et $\mathcal{N}lf \geq 1$.

dépendances de données :

$$\mathcal{E}xcl = \left\{ \| (o_1 \triangleright s_{1,2})^1, \; \ldots, \; (o_1 \triangleright s_{1,2})^{\mathcal{N}lf+1} \| \right\}$$

où $(o_1 \triangleright s_{1,2})^k$ représente la k^{ieme} réplique de $(o_1 \triangleright o_2)$.

6.3.4 Tolérance aux fautes des processeurs et des liens de communication

Dans cette section, nous proposons une nouvelle technique de transformation qui permet de combiner les deux transformations précédentes afin de tolérer $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}lf$ fautes de liens de communication. Cette transformation de graphe d'algorithme se fait par AAA-TP en deux étapes : une étape de réplication et une étape de distribution.

6.3.4.1 Étape de réplication

Cette étape de réplication consiste à répliquer tous les composants logiciels dans le but de tolérer $\mathcal{N}pf$+$\mathcal{N}lf$ fautes. Cette réplication se fait en deux temps.

Dans un premier temps, afin de tolérer les fautes d'au plus $\mathcal{N}pf$ processeurs, il est nécessaire de placer chaque opération de $\mathcal{A}lg$ sur $\mathcal{N}pf$+1 processeurs de $\mathcal{A}rc$. Donc, pour la réplication des opérations de $\mathcal{A}lg$, nous utilisons le même schéma de transformation que celui présenté dans la figure 6.4b. Par exemple, dans la figure 6.7, les deux opérations o_1 et o_2, de la figure 6.4a, sont répliquées en $\mathcal{N}pf$+1 répliques dans $\mathcal{A}lg^*$, respectivement l'ensemble $\mathcal{R}ep(o_1)$ et l'ensemble $\mathcal{R}ep(o_2)$. En plus, chaque réplique o_2^k doit être équipée d'une opération de contrôle $s_{1,2}^k$, ce qui ajoute à $\mathcal{A}lg^*$ un ensemble $\mathcal{R}ep(s_{1,2})$ de $\mathcal{N}pf$+1 opérations de contrôle.

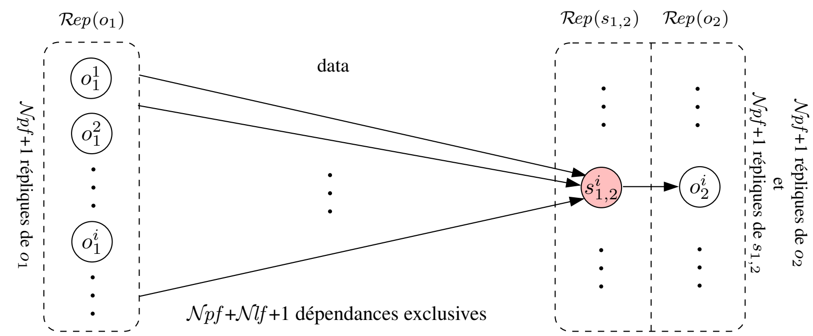


FIG. 6.7 – Transformation du graphe d'algorithme dans le cas où $\mathcal{N}pf \geq 0$ et $\mathcal{N}lf \geq 0$.

Dans un deuxième temps, étant donné qu'une dépendance de données peut être placée sur une route de communication, composée de plusieurs composants matériels (processeurs et liens de communication), et qu'une faute temporelle active d'un de ces composants peut provoquer une défaillance du système, chaque opération de contrôle $s_{i,j}$ doit recevoir ses données d'entrées de $\mathcal{R}ep(o_i)$ via $\mathcal{N}pf$+$\mathcal{N}lf$+1 routes disjointes pour masquer les erreurs provoquées par ces fautes. La dépendance de données $(o_i \triangleright o_j)$ est donc répliquée dans $\mathcal{A}lg^*$ en $\mathcal{N}pf$+$\mathcal{N}lf$+1 dépendances exclusives de données entre l'ensemble $\mathcal{R}ep(o_i)$ et chaque opération de contrôle $s_{i,j}^k$ de o_j^k, réplique de o_j et successeur de o_i. Elle est aussi répliquée entre chaque réplique o_j^k et son opération de contrôle $s_{i,j}^k$.

Par exemple, dans la figure 6.7, la dépendance $(o_1 \triangleright o_2)$ de la figure 6.4a est répliquée en $\mathcal{N}pf$+$\mathcal{N}lf$+1 répliques entre $\mathcal{R}ep(o_1)$ et $\mathcal{R}ep(s_{1,2})$, et en une réplique entre chaque réplique o_2^k et chaque opération de contrôle $s_{1,2}^k$.

6.3.4.2 Étape de distribution

La dernière étape de la transformation du graphe d'algorithme $\mathcal{A}lg$ consiste à connecter ces $\mathcal{N}pf$+$\mathcal{N}lf$+1 dépendances exclusives entre les $\mathcal{N}pf$+1 répliques $\mathcal{R}ep(o_i)$ de o_i et chaque opération de contrôle $s_{i,j}^k$. Ces connections doivent tolérer n'importe quelle combinaison arbitraire d'au plus $\mathcal{N}pf$ fautes de processeurs et d'au plus $\mathcal{N}lf$ fautes de liens de communication. Deux difficultés se posent : d'une part l'architecture n'est pas forcément complètement connectée, et d'autre part une route peut défaillir à cause soit d'une faute d'un lien de communication, soit d'une faute d'un processeur servant au routage. Pour résoudre ce problème, nous proposons une distribution de ces dépendances qui est *moins coûteuse en nombre de dépendances de données*, et donc moins coûteuse en nombre de communications. Afin de bien expliquer notre technique de distribution des dépendances $\mathcal{R}ep(o_i \triangleright o_j)$ entre les opérations de $\mathcal{R}ep(o_i)$ et chaque opération de contrôle de $\mathcal{R}ep(s_{i,j})$, nous présentons tout d'abord ses principes dans le cas particulier où $\mathcal{N}pf$=1 et $\mathcal{N}lf$=1, pour le graphe d'algorithme $\mathcal{A}lg$ de la figure 6.4a.

La figure 6.8a représente le nouveau graphe d'algorithme $\mathcal{A}lg^*$ obtenu en appliquant les transformations de la première étape de réplication sur le graphe de la figure 6.4a. $\mathcal{A}lg^*$ est donc composé de :

- trois ensembles $\mathcal{R}ep(o_1) = \{o_1^k;\ k \in [1..2]\}$ (répliques de o_1), $\mathcal{R}ep(o_2) = \{o_2^k;\ k \in [1..2]\}$ (répliques de o_2) et $\mathcal{R}ep(s_{1,2}) = \{s_{1,2}^k;\ k \in [1..2]\}$ (opérations de contrôle) ;
- une dépendance de données ($s_{1,2}^k \rhd o_2^k$), réplique de ($o_1 \rhd o_2$), entre chaque k^{ieme} opération de $\mathcal{R}ep(s_{1,2})$ et chaque k^{ieme} opération de $\mathcal{R}ep(o_2)$;
- trois dépendances de données exclusives, répliques de ($o_1 \rhd o_2$), entre l'ensemble $\mathcal{R}ep(o_1)$ et chaque opération de $\mathcal{R}ep(s_{1,2})$.

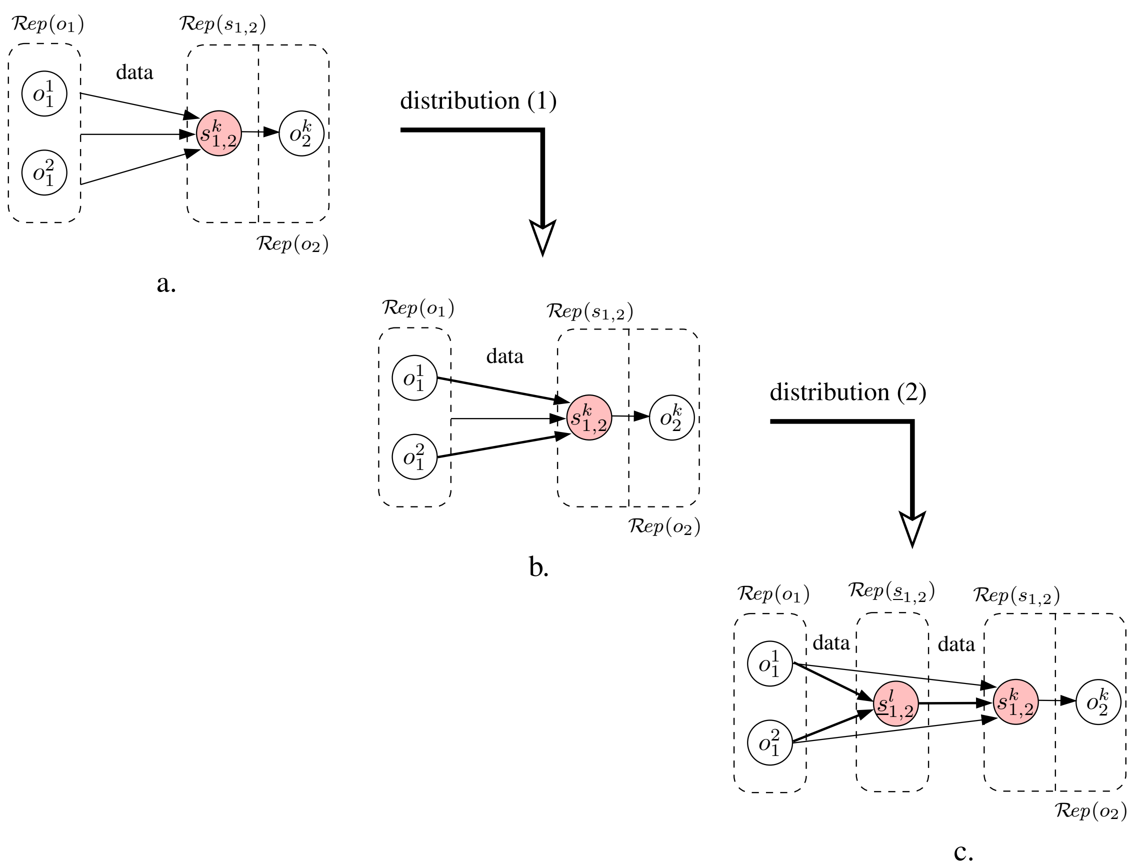


FIG. 6.8 – Schéma de transformation de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 1$.

Il ne nous reste plus qu'à connecter les trois dépendances exclusives de chaque $s_{1,2}^k$ aux deux répliques o_1^1 et o_1^2 de $\mathcal{R}ep(o_1)$. Pour cela, nous connectons, dans un première temps, chaque réplique de o_1 à une seule dépendance parmi ces trois dépendances, comme cela est montré sur la figure 6.8b. Dans un deuxième temps, nous remplaçons la troisième dépendance (celle qui n'est pas encore connectée) par une nouvelle opération de contrôle $\underline{s}_{1,2}^l$. Ensuite, nous relions toutes les opérations de $\mathcal{R}ep(o_1)$ à cette nouvelle opération $\underline{s}_{1,2}^l$, qui est elle aussi reliée à la k^{ieme} opération de $\mathcal{R}ep(s_{1,2})$, comme cela est montré sur la figure 6.8b. Enfin, la figure 6.9 représente le nouveau graphe d'algorithme $\mathcal{A}lg^*$ du graphe de la figure 6.4a.

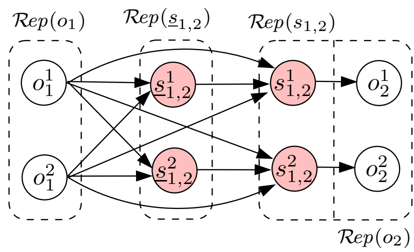


FIG. 6.9 – Schéma de transformation final de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf = 1$ et $\mathcal{N}lf = 1$.

Remarque 13 *Nous notons cette nouvelle opération de contrôle par $\underline{s}_{i,j}$, au lieu de $s_{i,j}$, ceci pour la différencier des anciennes opérations de contrôle, puisque, en plus des deux rôles de vérification et de sélection, elle réalise une opération de routage. Elle sert aussi à créer une troisième source ($\mathcal{N}pf$+$\mathcal{N}lf$+1=3) pour fournir la dépendance de donnée ($o_1 \triangleright o_2$) à son unique successeur $s_{1,2}^k$. Ces trois sources sont ainsi les opérations o_1^1, o_1^2 et $\underline{s}_{1,2}^l$.*

Cette transformation génère une liste de relations d'exclusion $\mathcal{E}xcl$ entre opérations et aussi entre dépendances de données. Par exemple, pour la figure 6.9 :

$$\mathcal{E}xcl = \bigcup_{i=1}^{2} \{\|o_i^1, o_i^2\|\} \bigcup_{k=1}^{2} \{\|(o_1^1 \triangleright \underline{s}_{1,2}^k), (o_1^2 \triangleright \underline{s}_{1,2}^k)\|\} \bigcup_{k=1}^{2} \{\|(o_1^1 \triangleright s_{1,2}^k), (o_1^2 \triangleright s_{1,2}^k), (\underline{s}_{1,2}^k \triangleright s_{1,2}^k)\|\}$$

Ce schéma de transformation peut être réduit au niveau de l'heuristique de distribution/ordonnancement :

- Soit en connectant la première réplique $s_{1,2}^1$ à $s_{1,2}^2$, et en supprimant la nouvelle opération de contrôle $\underline{s}_{1,2}^2$ de $\mathcal{A}lg^*$ et ses dépendances, comme cela est montré sur la figure 6.10a.

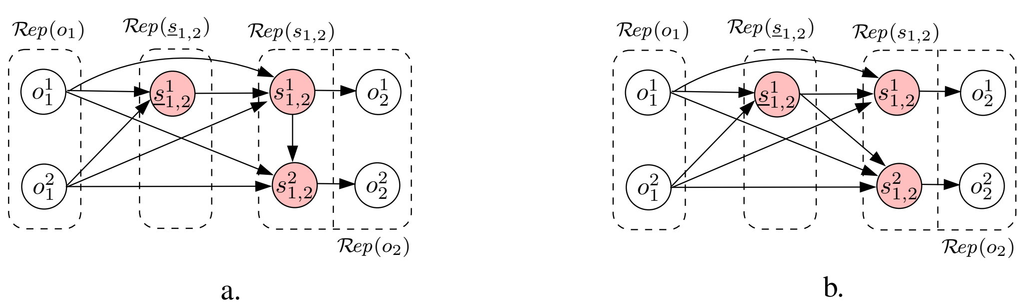


FIG. 6.10 – Schémas de transformation réduits.

Dans ce cas, l'ensemble $\mathcal{E}xcl$ devient :

$$\begin{aligned}\mathcal{E}xcl &= \bigcup_{i=1}^{2}\{\|o_i^1, o_i^2\|\} \cup \{\|(o_1^1 \triangleright \underline{s}_{1,2}^1), (o_1^2 \triangleright \underline{s}_{1,2}^1)\|\} \bigcup \{\|(o_1^1 \triangleright s_{1,2}^1), (o_1^2 \triangleright s_{1,2}^1), (\underline{s}_{1,2}^1 \triangleright s_{1,2}^1)\|\} \\ &\bigcup \{\|(o_1^1 \triangleright s_{1,2}^2), (o_1^2 \triangleright s_{1,2}^2), (\underline{s}_{1,2}^1 \triangleright s_{1,2}^1 \triangleright s_{1,2}^2)\|\}\end{aligned}$$

où $(\underline{s}_{1,2}^1 \triangleright s_{1,2}^1 \triangleright s_{1,2}^2)$ désigne les deux dépendances de données $(\underline{s}_{1,2}^1 \triangleright s_{1,2}^1)$ et $(s_{1,2}^1 \triangleright s_{1,2}^2)$ qui doivent être placées en série sur une même route de communication.

- Soit en utilisant $\underline{s}_{1,2}^1$ au lieu de $\underline{s}_{1,2}^2$, comme cela est montré sur la figure 6.10b. Dans ce cas, l'ensemble $\mathcal{E}xcl$ devient :

$$\begin{aligned}\mathcal{E}xcl &= \bigcup_{i=1}^{2}\{\|o_i^1, o_i^2\|\} \cup \{\|(o_1^1 \triangleright \underline{s}_{1,2}^1), (o_1^2 \triangleright \underline{s}_{1,2}^1)\|\} \bigcup \{\|(o_1^1 \triangleright s_{1,2}^1), (o_1^2 \triangleright s_{1,2}^1), (\underline{s}_{1,2}^1 \triangleright s_{1,2}^1)\|\} \\ &\bigcup \{\|(o_1^1 \triangleright s_{1,2}^2), (o_1^2 \triangleright s_{1,2}^2), (\underline{s}_{1,2}^1 \triangleright s_{1,2}^2)\|\}\end{aligned}$$

- Dans la mesure où le coût d'une communication intra-processeur est négligeable, la réplication en plus de $\mathcal{N}pf$+1 copies de certaines opérations de $\mathcal{A}lg$ peut réduire le temps global de communication. Par exemple, si dans la figure 6.9 une troisième réplique o_1^3 de o_1 est placée sur le même processeur que la réplique o_2^2, alors o_2^2 peut recevoir ses données d'entrée via une communication intra-processeur $(o_1^3 \triangleright o_2^2)$. Dans ce cas, les opération $s_{1,2}^2$ et $\underline{s}_{1,2}^2$ et leurs dépendances de données peuvent être remplacées par une réplique o_1^3 de o_1 et une dépendance de données $(o_1^3 \triangleright o_2^2)$, comme cela est montré sur la figure 6.11a. Ce principe a été déjà utilisé par plusieurs algorithmes de distribution/ordonnancement dans le but de minimiser la longueur d'une telle distribution/ordonnancement [2].

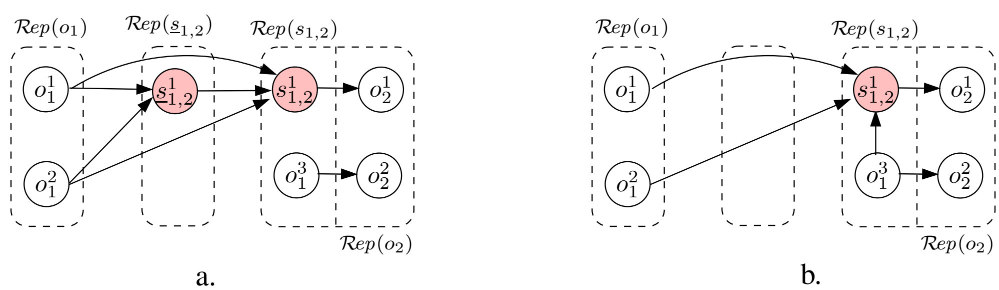


FIG. 6.11 – Réplication de certaines opérations en plus de $\mathcal{N}pf$+1 copies.

Dans ce cas, l'ensemble $\mathcal{E}xcl$ devient :

$$\begin{aligned}\mathcal{E}xcl &= \{\|o_1^1, o_1^2, o_1^3\|\} \cup \{\|o_2^1, o_2^2\|\} \bigcup \{\|(o_1^1 \triangleright \underline{s}_{1,2}^1), (o_1^2 \triangleright \underline{s}_{1,2}^1)\|\} \\ &\cup \{\|(o_1^1 \triangleright s_{1,2}^1), (o_1^2 \triangleright s_{1,2}^1), (\underline{s}_{1,2}^1 \triangleright s_{1,2}^1)\|\}\end{aligned}$$

Ce schéma peut être aussi réduit en remplaçant l'opération $\underline{s}_{1,2}^1$ et ses dépendances de données par une seule dépendance $(o_1^3 \triangleright s_{1,2}^1)$ entre o_1^3 et $s_{1,2}^1$, comme cela est montré sur la figure 6.11b.

Dans ce cas, l'ensemble $\mathcal{E}xcl$ devient :

$$\mathcal{E}xcl \;=\; \left\{\|o_1^1, o_1^2, o_1^3\|\right\} \cup \left\{\|o_2^1, o_2^2\|\right\} \cup \left\{\|(o_1^1 \rhd s_{1,2}^1), (o_1^2 \rhd s_{1,2}^1), (o_1^3 \rhd s_{1,2}^1)\|\right\}$$

Enfin, dans le cas général, où $\mathcal{N}pf \geq 1$ et $\mathcal{N}lf \geq 1$, le schéma de transformation est montré sur la figure 6.12.

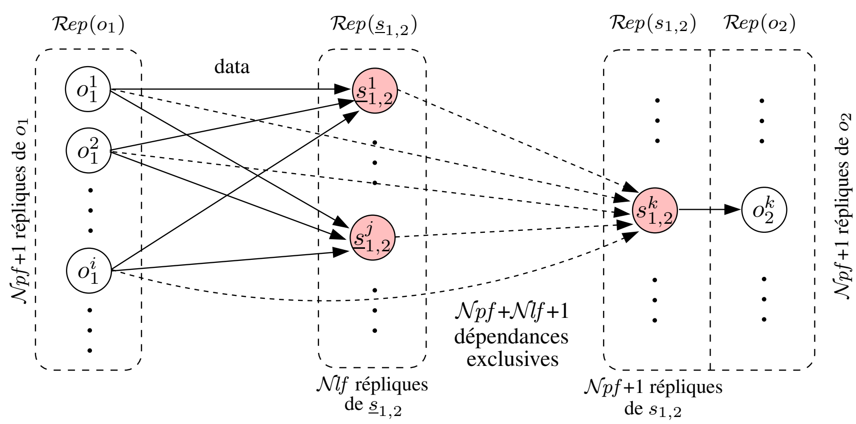


FIG. 6.12 – Schéma de transformation final de $\mathcal{A}lg$ en $\mathcal{A}lg^*$ pour $\mathcal{N}pf \geq 1$ et $\mathcal{N}lf \geq 1$.

6.4 Phase d'adéquation : heuristique de distribution et d'ordonnancement

Après la phase d'initialisation, où toutes les opérations et les dépendances de données du graphe d'algorithme $\mathcal{A}lg$ sont répliquées en un nouveau graphe d'algorithme $\mathcal{A}lg^*$, la deuxième phase d'adéquation est basée sur une heuristique de distribution/ordonnancement hors-ligne, qui consiste à mettre en correspondance de manière efficace le nouveau graphe d'algorithme $\mathcal{A}lg^*$ sur le graphe d'architecture $\mathcal{A}rc$. La distribution/ordonnancement générée par cette heuristique est statique [3], c'est-à-dire que les opérations (resp. communications) placées sur chaque processeur (resp. lien) sont toutes ordonnées avant la mise en exploitation du système, et cet ordre ne change pas en cours d'exécution du système, ce qui permet de vérifier hors-ligne la contrainte temps réel $\mathcal{R}tc$.

Les tâches principales de cette heuristique sont :

- *Placement actif :* puisque nous avons choisi le mécanisme de masquage des erreurs pour tolérer des fautes matérielles, toutes les opérations de calcul, d'entrée/sortie, de mémoire et de contrôle doivent être exécutées une seule fois durant chaque cycle d'exécution de l'algorithme $\mathcal{A}lg^*$ sur l'architecture $\mathcal{A}rc$. Ainsi, toutes les données des dépendances de données de $\mathcal{A}lg^*$ doivent être envoyées sur des routes de communication de $\mathcal{A}rc$ en respectant l'ensemble $\mathcal{E}xcl$.

- *Contraintes :* l'heuristique doit prendre en compte les coûts d'exécution $\mathcal{E}xe$, la contrainte temps réel $\mathcal{R}tc$, les contraintes matérielles $\mathcal{D}is$ et les contraintes d'exclusion $\mathcal{E}xcl$.
- *Calcul des dates de début d'exécution :* à chaque composant logiciel est associée une date de début d'exécution en absence de fautes, notée S_{best}, et aussi une date de début d'exécution en présence de $\mathcal{N}pf$+$\mathcal{N}lf$ fautes, notée S_{worst}. Donc, cette heuristique doit calculer hors-ligne ces deux dates pour chaque composant logiciel.
- *Prévision du comportement temps réel :* le calcul des deux dates de début d'exécution de chaque composant logiciel nous permet de vérifier hors-ligne si la contrainte temps réel est respectée ou non.

6.4.1 Notations

En plus des notations définies dans la section 2.6.2 (page 28), nous donnons dans cette section d'autres notations qui seront utilisées dans le reste de ce travail. L'heuristique que nous proposons est une modification de celles présentées dans [29, 31, 32], donc certaines notations sont les mêmes, alors que d'autres ont dû être adaptées à un graphe d'algorithme avec redondances :

- $\mathcal{O}p(p_j)$: la liste des opérations déjà placées sur le processeur p_j.
- $\mathcal{C}om(l)$: la liste des dépendances de données déjà placée sur le lien l.
- $X^{(n)}$: l'exposant n entre parenthèses désigne l'étape de l'heuristique, c'est-à-dire après avoir placé la n^{ieme} opération ; donc, $X^{(n)}$ désigne X à l'étape n de l'heuristique ;
- $Et^{(n)}_{exc}(o_i, p_j)$: la date de fin d'exécution de l'opération o_i placée sur le processeur p_j,
- $Et^{(n)}_{com}(o_i \triangleright o_k)$: la date de fin de communication entre o_i et o_k,
- $St^{(n)}_{best}(o_i, p_j)$: la date de début au-plus-tôt depuis le début de o_i sur p_j, ne prenant en compte que la première réplique de chaque opération de ses prédécesseurs.
- $St^{(n)}_{worst}(o_i, p_j)$: la date de début au-plus-tôt depuis le début de o_i sur p_j, prenant en compte toutes les répliques de ses prédécesseurs.
- $O_{succ}(o_i)$: définit l'ensemble $\mathcal{R}ep(o_j)$ des successeurs de $\mathcal{R}ep(s_{i,j})$, avec $\mathcal{R}ep(s_{i,j})$ est l'ensemble des opérations de contrôle successeurs de $\mathcal{R}ep(o_i)$, .
- $O_{pred}(o_j)$: définit l'ensemble $\mathcal{R}ep(o_i)$ des prédécesseurs de $\mathcal{R}ep(s_{i,j})$, avec $\mathcal{R}ep(s_{i,j})$ est l'ensemble des opérations de contrôle prédécesseurs de $\mathcal{R}ep(o_j)$.

6.4.2 Principes de l'heuristique

L'heuristique de distribution/ordonnancement que nous proposons est un algorithme par *construction progressive* de type *glouton* (greedy) [11]. Elle est basée sur une fonction de coût appelée la *nouvelle pression d'ordonnancement*, notée $\tilde{\sigma}^{(n)}_{o_i,p_j}$, dont l'objectif global est de minimiser la longueur du chemin critique en absence et en présence d'au plus $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances. Cette nouvelle fonction de la pression d'ordonnancement est une adaptation de la fonction de la pression d'ordonnancement, donnée par l'equation (2.5), aux graphes d'algorithmes avec redondance.

Avant de présenter cette fonction, nous présentons tout d'abord les principes de placement actif des opérations et des dépendances de données du nouveau graphe d'algorithme $\mathcal{A}lg^*$ sur le graphe d'architecture $\mathcal{A}rc$:

- chaque opération envoie les données de ses dépendances exclusives en parallèle à toutes ses opérations successeurs ;
- chaque opération de contrôle $\underline{s}^m_{i,j}$ reçoit ses données d'entrée en $\mathcal{N}pf$+1 exemplaires, et dès qu'elle reçoit le premier exemplaire, elle le renvoie à son unique successeur $s^k_{i,j}$ via une communication de routage, puis elle ignore les autres exemplaires ;
- chaque opération de contrôle $s^k_{i,j}$ reçoit ses données d'entrée en $\mathcal{N}pf$+$\mathcal{N}lf$+1 exemplaires, et dès qu'elle reçoit le premier exemplaire, elle le renvoie à son unique successeur o^k_j via une communication intra-processeur, puis elle ignore les autres exemplaires ;
- chaque opération o^k_i de calcul, d'entrée/sortie et de mémoire reçoit ses données d'entrée en un seul exemplaire via une communication intra-processeur ;
- à chaque opération de $\mathcal{A}lg^*$ sont associées deux dates de début d'exécution : la date où l'opération reçoit le premier exemplaire de ses données d'entrée, appelée St_{best}, et la date où l'opération reçoit le dernier exemplaire de ses données d'entrée, appelée St_{worst}. Par exemple, dans la distribution/ordonnancement de la figure 6.13, St_{best} de l'opération $s^2_{1,2}$ est la date de la fin de la communication des données de $(o^1_1 \triangleright s^2_{1,2})$ sur le lien l_{12}, et St_{worst} de l'opération $s^2_{1,2}$ est la date de fin de la communication des données de $(\underline{s}^2_{12} \triangleright s^2_{1,2})$ sur le lien l_{24}. Dans cette distribition/ordonnancement, les opérations (resp. les communications) sont représentées par des boîtes dont la hauteur est proportionnelle à leur durée d'exécution (resp. de communication).

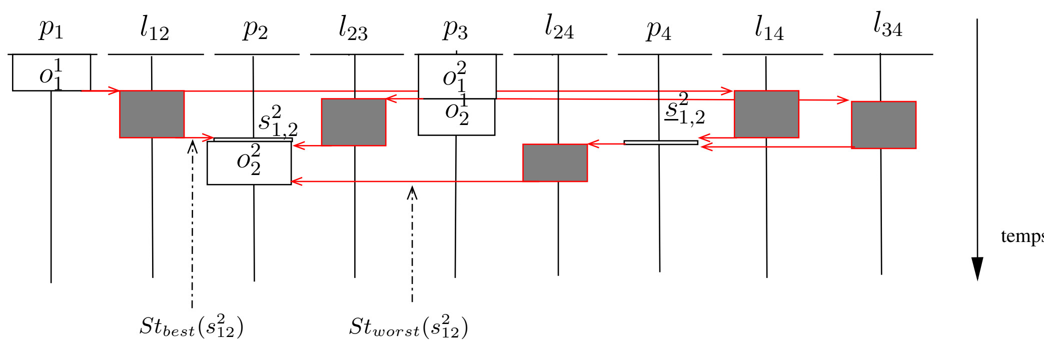


FIG. 6.13 – Exemple d'une distribution/ordonnancement.

Dans certains cas, une opération de contrôle $s^m_{i,j}$ peut recevoir ses données d'entrée une seule fois via une communication intra-processeur ; c'est le cas où elle est placée sur un processeur implantant une réplique o^k_i de ses prédécesseurs. Dans ce cas, dans un premier temps, les dépendances de données $(o^*_i \triangleright s^m_{i,j})$ et les opérations de contrôle $\mathcal{R}ep(\underline{s}_{i,j})$, prédécesseurs de $s^m_{i,j}$, seront supprimées du $\mathcal{A}lg^*$, et donc o^k_i devient la seule réplique de o_i prédécesseur de $s^m_{i,j}$. Et puisque $s^m_{i,j}$ a un seul successeur o^m_j, l'opération $s^m_{i,j}$ et ses deux dépendances de données $(o^k_i \triangleright s^m_{i,j})$ et $(s^m_{i,j} \triangleright o^m_j)$ seront remplacées dans $\mathcal{A}lg^*$ par une seule dépendance de données $(o^k_i \triangleright o^m_j)$ entre o^k_i et o^m_j. Par exemple, dans la distribution/ordonnancement de la figure 6.13, les opérations $\underline{s}^1_{1,2}$ et $s^1_{1,2}$ sont supprimées du graphe $\mathcal{A}lg^*$, et leurs dépendances de données sont toutes remplacées par une seule dépendance $(o^2_1 \triangleright o^1_2)$, qui est ensuite implantée comme une communication intra-processeur.

Les deux dates St_{best} et St_{worst} sont calculées de la façon suivante :

- Pour chaque opération o_i placée sur un processeur p et pour chaque dépendance de données $(o_i \triangleright o_j)$ placée sur un lien l :

$$\begin{cases} St^{(n)}_{best}(o_i \triangleright o_j, l) = \max\left(St^{(n)}_{best}(o_i, p) \quad , \max_{(o_k \triangleright o_m) \in \mathcal{C}om(l)} St^{(n)}_{best}(o_k \triangleright o_m, l)\right) \\ St^{(n)}_{worst}(o_i \triangleright o_j, l) = \max\left(St^{(n)}_{worst}(o_i, p) \quad , \max_{(o_k \triangleright o_m) \in \mathcal{C}om(l)} St^{(n)}_{worst}(o_k \triangleright o_m, l)\right) \end{cases} \tag{6.1}$$

Rappelons que $\mathcal{C}om(l)$ désigne l'ensemble des communications déja placées sur le lien l, et donc exécutées avant la communication $(o_i \triangleright o_j)$.

$$\begin{cases} St^{(n)}_{best}(o_i, p) = \max\left(\max_{o_j \in pred(o_i)} \left(\min_k St^{(n)}_{best}(o^k_j \triangleright o_i) \quad , \max_{o \in \mathcal{O}p(p)} St^{(n)}_{best}(o, p)\right)\right) \\ St^{(n)}_{worst}(o_i, p) = \max\left(\min_{o_j \in pred(o_i)} \left(\max_k St^{(n)}_{worst}(o^k_j \triangleright o_i) \quad , \max_{o \in \mathcal{O}p(p)} St^{(n)}_{worst}(o, p)\right)\right) \end{cases} \tag{6.2}$$

Rappelons que $\mathcal{O}p(p)$ désigne l'ensemble des opérations déja placées sur le processeur p, et donc exécutées avant l'opération o_i.

Enfin, puisque chaque opération de $\mathcal{A}lg^*$ possède deux dates de début d'exécution, St_{best} en absence de défaillance et St_{worst} en présence de $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances, et afin de minimiser la longueur de la distribution/ordonnancement en absence et en présence de défaillances, nous avons choisi de remplacer la date St de la fonction de la pression d'ordonnancement de l'equation (2.5) par St_{worst}, et donc l'équation de la nouvelle pression d'ordonnancement devient :

$$\tilde{\sigma}^{(n)}(o_i, p_j) := S^{(n)}_{worst}(o_i, p_j) + \overline{S}^{(n)}(o_i) - R^{(n-1)} \tag{6.3}$$

6.4.3 Présentation de l'heuristique

L'heuristique de distribution/ordonnancement consiste à placer et à ordonnancer, à chaque étape (n), plusieurs opérations (resp. dépendances de données) sur plusieurs processeurs (resp. liens). L'algorithme de l'heuristique se divise en cinq phases : une phase d'initialisation, une phase de vérification des routes disjointes, une phase de sélection, une phase de distribution/ordonnancement, puis une phase de mise à jour :

ALGORITHME

- ***Entrées =*** *Graphe d'algorithme transformé $\mathcal{A}lg^*$, graphe d'architecture $\mathcal{A}rc$, relations d'exclusion $\mathcal{E}xcl$, caractéristiques d'exécution $\mathcal{E}xe$, contrainte temps réel $\mathcal{R}tc$, et contraintes matérielles $\mathcal{D}is$;*
- ***Sortie =*** *Allocation spatiale et temporelle de $\mathcal{A}lg$ sur $\mathcal{A}rc$, prédictible et tolérante à $\mathcal{N}pf$ fautes de processeurs et à $\mathcal{N}lf$ fautes de liens de communication ;*

INITIALISATION

Initialiser la liste des opérations candidates, et la liste des opérations déjà placées :

$O_{cand}^{(1)} := \{$*opérations de* $\mathcal{A}lg^*$ *sans prédécesseurs*$\}$;

$O_{fin}^{(1)} := \emptyset$;

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

Tant que $O_{cand}^{(n)} \neq \emptyset$ ***faire***

- *Mettre la première réplique de chaque opération de $O_{cand}^{(n)}$ dans $O_{first}^{(n)}$*

  $O_{first}^{(n)} := \{o_j^1 \mid o_j^1 \in O_{cand}^{(n)}\}$;

VÉRIFICATION DES ROUTES DISJOINTES

- ***Si*** $O_{fin}^{(n)} \neq \emptyset$ ***Alors***
  - ***Pour chaque*** *opération de contrôle $s_{i,j}^k$ de $o_j^1 \in O_{first}^{(n)}$* ***faire***
    - *Calculer l'ensemble $\mathcal{P}(o_j^1)$ des processeurs, tels que pour chaque $p \in \mathcal{P}$ et chaque $o_i^m \in O_{fin}^{(n)}$ prédécesseur de $s_{i,j}^k$:*
      - *Soit o_j^1 et $s_{i,j}^k$ peuvent être placées sur le même processeur que o_i^m ;*
      - *Soit il existe selon $\mathcal{E}xcl$ et $\mathcal{D}is$:*
        - *$\mathcal{N}lf$ processeurs ($\mathcal{P}$') pour placer les opérations de contrôle $\mathcal{R}ep(\underline{s}_{i,j})$ de $s_{i,j}^k$, et*
        - *$\mathcal{N}pf$+$\mathcal{N}lf$+1 routes disjointes reliant $\mathcal{P}$' et les processeurs de $\mathcal{R}ep(o_i)$ à p ;*
    - ***Si*** *le nombre de processeurs dans $\mathcal{P}(o_i^1)$ est inférieur à $\mathcal{N}pf$+1*

      Alors return *« impossible de trouver des routes disjointes » ;*
  - ***Fin pour chaque***

SÉLECTION

- *Sélectionner pour chaque opération candidate o_i^1 de $O_{first}^{(n)}$ un ensemble $\mathcal{P}_{best}$ de $\mathcal{N}pf$+1 processeurs de $\mathcal{P}$ qui minimise la nouvelle fonction de la pression d'ordonnancement (équation 6.3) ;*
- *Sélectionner, parmi les processeurs de $\mathcal{P}_{best}(o_i^1)$, le meilleur processeur p_{best} qui maximise la nouvelle fonction de la pression d'ordonnancement (équation 6.3) pour chaque opération candidate o_i^1 de $O_{first}^{(n)}$;*
- *Sélectionner, parmi les couples (o_i^1, p_{best}), le meilleur couple (o_{best}^1, p_{best}) qui maximise la nouvelle fonction de la pression d'ordonnancement (équation 6.3) ;*

DISTRIBUTION ET ORDONNANCEMENT

- ***Pour chaque*** *couple* (o^k_{best}, p^k_{best}) ***faire***
  - ***Pour chaque*** $s^k_{i,best}$ *prédécesseur de* o^k_{best} ***faire***
    - *Soient* $\mathcal{R}ep(o_i)$ *et* $\mathcal{R}ep(\underline{s}_{i,best})$ *les prédécesseurs de* $s^k_{i,best}$ *;*
    - ***Si*** *il existe une réplique* o^m_i *placée sur le processeur* p^k_{best}
      Alors *Supprimer toutes les opérations de* $\mathcal{R}ep(\underline{s}_{i,best})$ *et l'opération* $s^k_{i,best}$*, ainsi que leurs dépendances du graphe* $\mathcal{A}lg^*$*, et ajouter la dépendance* ($o^m_i \triangleright o^k_{best}$) *à* $\mathcal{A}lg^*$*, ensuite, placer et ordonnancer cette dépendance comme une opération de communication intra-processeur ;*
    - ***Sinon***
      - ***Pour chaque*** $s \in \{\mathcal{R}ep(s^k_{i,best}) \cup s^k_{i,best}\}$ *et en respectant* $\mathcal{D}is$ *et* $\mathcal{E}xcl$ ***faire***
        - *Sélectionner le meilleur processeur* p_s *pour* s *qui minimise l'équation 6.3 ;*
        - *Placer et ordonnancer les opérations de communication induites par cette sélection en utilisant l'équation 6.1 ;*
        - *Calculer* $St^{(n)}_{best}(s, p_s)$ *et* $St^{(n)}_{worst}(s, p_s)$ *(équation 6.2) ;*
        - *Placer et ordonnancer* s *sur* p^k_{best} *à* $St^{(n)}_{best}(s, p_s)$ *;*
      - ***Fin pour chaque***
  - ***Fin pour chaque***
  - *Placer et ordonnancer toutes les dépendances* ($s^k_{i,best} \triangleright o^k_{best}$) *comme des communications intra-processeur ;*
  - *Calculer* $St^{(n)}_{best}(o^k_{best}, p_{best})$ *et* $St^{(n)}_{worst}(o^k_{best}, p_{best})$ *(équation 6.2) ;*
  - *Placer et ordonnancer* o^k_{best} *sur* p^k_{best} *à* $St^{(n)}_{best}(o^k_{best}, p_{best})$ *;*
- ***Fin pour chaque***
- *Si c'est possible appliquer les principes de transformation des figures 6.10 et 6.11 ;*

MISE À JOUR

- *Mettre à jour la liste des opérations candidates et déjà placées :*
  $O^{(n+1)}_{fin} := O^{(n)}_{fin} \cup \mathcal{R}ep(o_{best})$;
  $O^{(n+1)}_{cand} := O^{(n)}_{cand} - \mathcal{R}ep(o_{best}) \cup \left\{ o \in O_{succ}(o^1_{best}) \mid O_{pred}(o) \subseteq O^{(n+1)}_{fin} \right\}$;

Fin tant que

FIN DE L'ALGORITHME

6.4.3.1 Phase d'initialisation

Cette phase consiste à initialiser la liste des opérations candidates $O^{(1)}_{cand}$ avec des opérations sans prédécesseurs. Donc, les seules opérations implantables à cette première étape de l'heuristique sont les opérations d'entrée (capteurs). Cette liste des candidates ne peut contenir que les opérations répliques des opérations de $\mathcal{A}lg$, puisque à chaque étape (n) de l'heuristique nous plaçons $\mathcal{N}pf$+1 opérations candidates de $\mathcal{R}ep(o_j)$ ainsi que leurs opérations de contrôle $\mathcal{R}ep(s_{i,j})$ et $\mathcal{R}ep(\underline{s}_{i,j})$. Enfin, la liste des opérations déjà placées $O_{fin}^{(1)}$ est vide à cette étape.

6.4.3.2 Phase de vérification des routes disjointes

L'objectif principal de cette phase est de calculer l'ensemble des processeurs $\mathcal{P}$ qui peuvent implanter chaque opération candidate o_j^k. Étant donné que chaque opération candidate o_j^k a plusieurs opérations de contrôle ($s_{i,j}^k$) qui doivent être placées sur le même processeur p que o_j^k, l'ensemble $\mathcal{P}$ est le même ensemble pour chaque opération $s_{i,j}^k$. De plus, étant donné que toutes les répliques $\mathcal{R}ep(o_j)$ sont identiques, il suffit de ne calculer cet ensemble $\mathcal{P}$ que pour la première réplique o_j^1, et donc uniquement pour les opérations de contrôle $\mathcal{R}ep(s_{i,j})$ de o_j^1.

Un processeur p est ajouté à la liste $\mathcal{P}(o_j^1)$ s'il vérifie les contraintes suivantes :

- il n'y a pas de contrainte de distribution $\mathcal{D}is$ de o_j sur p ;
- il existe pour chaque $\mathcal{R}ep(o_i)$ prédécesseurs de $s_{i,j}^1$ et de $\mathcal{R}ep(\underline{s}_{i,j})$:
  - soit une réplique $o_i^k \in \mathcal{O}p(p)$,
  - soit $\mathcal{N}lf$ processeurs ($\mathcal{P}$') qui peuvent implanter les opérations de $\mathcal{R}ep(\underline{s}_{i,j})$, et $\mathcal{N}pf$+$\mathcal{N}lf$+1 routes disjointes reliant $\mathcal{P}$' et les processeurs de $\mathcal{R}ep(o_i)$ à p.

Enfin, le résultat de cette phase est un ensemble $\mathcal{P}$ de processeurs candidats à implanter chaque opération de $O_{cand}^{(n)}$. Si pour une opération o_j^1 le nombre de processeurs de $\mathcal{P}(o_j^1)$ est inférieur à $\mathcal{N}pf$+1, alors l'heuristique échoue, à cause soit des contraintes de distribution, soit d'un nombre insuffisant de routes disjointes.

6.4.3.3 Phase de sélection

Dans cette phase, la candidate la plus urgente o_{best}^k est sélectionnée parmi toutes les opérations candidates pour être placée et ordonnancée. La règle de sélection choisie repose sur la nouvelle fonction de la pression d'ordonnancement donnée par l'équation 6.3. Le processus de sélection est réalisé en trois étapes :

- la première étape consiste à sélectionner pour chaque candidat o_j^1 un ensemble $\mathcal{P}_{best}$ de $\mathcal{N}pf$+1 processeurs parmi les processeurs de $\mathcal{P}(o_j^1)$ qui minimisent la fonction de coût de (6.3),
- puis, la deuxième étape consiste à sélectionner pour chaque candidat o_j^1 un processeur p_{best}, parmi les processeurs de $\mathcal{P}_{best}(o_j^1)$ qui maximise la fonction de coût de (6.3),
- enfin, la dernière étape consiste à sélectionner le couple le plus urgent (o_{best}^1,p_{best}) parmi tous les couples (o_j^1,p_{best}) qui maximisent la fonction de coût de (6.3).

6.4.3.4 Phase de distribution/ordonnancement

Après avoir sélectionné la meilleure candidate o_{best}^1 et ses $\mathcal{N}pf$+1 meilleurs processeurs $\mathcal{P}_{best}(o_{best}^1)$ pendant la phase de sélection, il ne reste plus qu'à le placer/ordonnancer sur son meilleur processeur p_{best}^1 à l'instant $St_{best}(o_{best}, p_{best}^1)$. Cependant, ses opérations de contrôle, qui sont ses prédécesseurs, ne sont pas encore placées/ordonnancées. Donc, l'heuristique sélectionne, place et ordonnance chaque opération $\mathcal{R}ep(s_{i,best})$ et $\mathcal{R}ep(\underline{s}_{i,best})$ sur son meilleur processeur en utilisant $\tilde{\sigma}$, St_{best} et St_{worst}. Il faut noter que si les données $(o_i \triangleright o_j)$ sont présentes sur le même processeur p^1_{best} avant que les opérations de contrôle ne soient placées/ordonnancées, alors l'opération o^1_{best} les utilise, et donc l'opération $s^1_{i,best}$ et ses prédécesseurs $\mathcal{R}ep(\underline{s}_{i,best})$ seront supprimées du graphe d'algorithme $\mathcal{A}lg^*$, ainsi que leurs dépendances (voir section 6.4.2). Enfin, la k^{ieme} opération de $\mathcal{R}ep(o_{best})$ est placée sur le k^{ieme} processeur de $\mathcal{P}_{best}$ de la même façon que o^1_{best}.

6.4.3.5 Phase de mise à jour

Cette phase consiste à mettre à jour tout d'abord la liste des opérations déjà placées et ensuite la liste des opérations candidates. Toutes les opérations de $\mathcal{R}ep(o_{best})$ sont supprimées de la liste des opérations candidates $O^{(n)}_{cand}$, et les nouvelles opérations ajoutées à cette liste sont les opérations de $\mathcal{A}lg^*$ qui ont tous leurs prédécesseurs dans la liste des opérations déjà placées. Cependant, suivant la structure de $\mathcal{A}lg^*$, la liste des nouvelles opérations n'est constituée que des opérations de contrôle qui ne peuvent pas devenir candidates à cause de notre stratégie de distribution/ordonnancement. Les nouvelles opérations candidates sont donc les prédécesseurs de toutes ces opérations de contrôle.

6.5 Prédiction du comportement temps réel

La seule contrainte à vérifier après l'étape de distribution/ordonnancement est la contrainte temps réel $\mathcal{R}tc$, c'est-à-dire que la durée d'exécution de l'algorithme sur l'architecture doit être inférieure à un seuil défini par $\mathcal{R}tc$. La vérification de cette contrainte temps réel est effectuée hors-ligne :

- *En absence de défaillance* : il suffit dans ce cas de vérifier si la date de la fin d'exécution $(St^{(n)}_{best}(o_{last}, p_j) + \mathcal{E}xe(o_{last}, p_j))$ de la dernière opération o_{last} de chaque processeur p_j est inférieure à $\mathcal{R}tc$. Sinon l'heuristique échoue à trouver une distribution/ordonnancement qui satisfasse cette contrainte temps réel.
- *En présence de $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances* : étant donné que chaque opération a une date de début d'exécution en présence de $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances, l'heuristique peut prédire la date de fin d'exécution de l'algorithme sur l'architecture en présence de $\mathcal{N}pf$+$\mathcal{N}lf$ défaillances. Cette date est égale à la date maximale de la fin d'exécution $(St^{(n)}_{worst}(o_{last}, p_j) + \mathcal{E}xe(o_{last}, p_j))$ de la dernière opération o_{last} de chaque processeur p_j. Elle doit être inférieure à $\mathcal{R}tc$, sinon l'heuristique échoue à trouver une solution valide qui satisfasse cette contrainte temps réel.

Dans le cas où la contrainte temps réel n'est pas satisfaite, le concepteur doit soit ajouter des composants matériels à son architecture $\mathcal{A}rc$, soit modifier ses contraintes, et ensuite réexécuter l'heuristique.

Remarque 14 *Rappelons que notre heuristique ne garantit pas de trouver toujours une solution valide, donc le fait qu'elle échoue ne signifie pas qu'il n'existe pas de solution valide, mais juste* *qu'elle n'a pas réussi à en trouver une.*

6.6 Extension de la méthodologie AAA-TP à la tolérance aux fautes des capteurs

Le capteur est un élément essentiel dans la conception des systèmes réactifs, car de son comportement dépend le bon fonctionnement de ces systèmes. Il est utilisé comme un moyen pour obtenir des informations pertinentes sur l'environnement que le système contrôle. En plus des fautes de processeurs et de liens de communication qui peuvent affecter ces systèmes, les fautes de capteurs sont inévitables, et les principales causes de ces fautes sont la température, la pression, les chocs, les vibrations, les champs magnétiques et l'humidité. Par exemple, « l'étude menée au sein de l'usine de ThyssenKrupp sur les causes des arrêts de leurs robots a identifié la principale cause de ces arrêts, qui était les capteurs. Plus précisément, les défaillances sont dues aux conditions d'environnements : chocs mécaniques et champs électromagnétiques » [60]. D'où la nécessité d'un mécanisme pour tolérer ces fautes.

La méthodologie AAA-TP que nous avons proposée dans ce chapitre suppose que les capteurs ne sont pas physiquement répliqués, c'est-à-dire que les processeurs implantant les répliques $\mathcal{R}ep(In_i)$ d'une même opération d'entrée In_i sont reliés à un seul capteur, que nous notons aussi In_i. La spécification des capteurs dans le modèle d'architecture (cf. section 4.3.1, page 56) était implicite : chaque opération d'entrée In_i du graphe d'algorithme $\mathcal{A}lg$ désigne l'existence d'un capteur In_i dans le graphe d'architecture $\mathcal{A}rc$, et ce capteur est connecté uniquement aux processeurs ayant une valeur $\mathcal{E}xe(In_i, p_j)$ différente de la valeur ∞. Par exemple, suivant le graphe d'algorithme de la figure 4.3 et le tableau 4.1, le graphe d'architecture de la figure 4.8 est constitué de deux capteurs In_1 et In_2, où In_1 (resp. In_2) est relié aux processeurs P_1 et P_2 (resp. P_1, P_2 et P_3), comme cela est montré sur la figure 6.14.

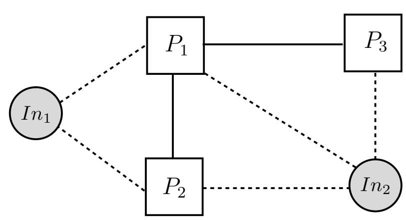


FIG. 6.14 – Modèle d'une architecture avec deux capteurs.

La méthodologie AAA-TP peut être facilement étendue pour tolérer aussi des fautes des capteurs, et ceci grâce aux nouvelles opérations de contrôle ajoutées au graphe d'algorithme initial $\mathcal{A}lg$. AAA-TP peut tolérer jusqu'à $\mathcal{N}sf$[4] fautes de capteurs, où les fautes peuvent être de deux types : fautes temporelles et fautes fonctionnelles :

[4] $\mathcal{N}sf$ = Number of Sensor Failures.

- **Fautes temporelles** : si on suppose que les capteurs sont à défaillances temporelles, c'est-à-dire que les valeurs des capteurs sont soit correctes et délivrées à temps, soit correctes et délivrées trop tôt, trop tard ou infiniment tard, alors $\mathcal{N}sf$+1 répliques de chaque capteurs sont nécessaire pour tolérer $\mathcal{N}sf$ fautes de capteurs [12],
- **Fautes fonctionnelles** : si on suppose que les capteurs sont à défaillances fonctionnelles, c'est-à-dire que les valeurs des capteurs sont soit correctes et délivrées à temps, soit non correctes et délivrées à temps, alors $2\mathcal{N}sf$+1 répliques de chaque capteurs sont nécessaire pour tolérer $\mathcal{N}sf$ fautes de capteurs [12].

Dans la suite de cette section, nous désignons par une faute de capteur soit une faute temporelle, soit une faute fonctionnelle, et nous désignons par le nombre S soit $\mathcal{N}sf$+1 (cas des fautes temporelles), soit $2\mathcal{N}sf$+1 (cas des fautes fonctionnelles). Ce modèle exclu les fautes byzantines.

6.6.1 Complexité du problème

Étant donné la complexité de concevoir un système tolérant à la fois à $\mathcal{N}pf$ fautes de processeurs, $\mathcal{N}lf$ fautes de liens de communication et $\mathcal{N}sf$ fautes de capteurs dans une architecture non complètement connectée, la plupart des solutions existantes dans la littérature supposent qu'au moins un de ces trois composants est fiable, c'est-à-dire sans faute. La complexité du problème s'explique dans le fait de :

- calculer le nombre N de répliques pour chaque opération d'entrée In_i afin de tolérer $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}sf$ fautes de capteurs. Pour tolérer $\mathcal{N}sf$ fautes de capteurs, il est nécessaire de répliquer physiquement chaque capteur In_i en S copies. Dans le cas général où N est différent de S, un problème complexe se présente : comment connecter les N répliques de chaque opération d'entrée In_i à leurs S capteurs physiques In_i, de telle sorte que cette connexion tolère $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}sf$ fautes de capteurs ?
- en outre, le problème sera plus compliqué si on considère les fautes des liens de communication. En effet, chaque opération de contrôle successeur d'une opération d'entrée doit faire, en plus d'une opération de vérification, une opération de vote, ce qui nécessite d'avoir plus de communication dans le système. Ainsi, le nombre de ces communications dépend de S, de $\mathcal{N}lf$ et du nombre de routes disjointes.

Enfin, de même que la plupart des solutions existantes dans la littérature, AAA-TP peut tolérer soit $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}sf$ fautes de capteurs, soit $\mathcal{N}lf$ fautes de liens de communication et $\mathcal{N}sf$ fautes de capteurs. Dans ces deux cas, il suffit de remplacer les opérations de contrôle par des voteurs [62, 12], et de réutiliser les transformations présentées dans les deux sections 6.3.2 et 6.3.3. Notons que dans ce cas, l'hypothèse 9 (la durée d'exécution d'une opération de contrôle est nulle) ne sera plus valable, et donc, il faudra calculer les coûts d'exécution $\mathcal{E}xe$ de chaque voteur utilisé sur chaque processeur du graphe d'architecture $\mathcal{A}rc$.

6.7 Simulations

Afin d'évaluer l'heuristique de distribution/ordonnancement de AAA-TP, nous avons comparé ses performances avec l'algorithme proposé par Hashimoto dans [39], appelé HBP (Height-Based Partitioning), qui est le plus proche du notre que nous avons trouvé dans la littérature. HBP est conçu pour des architectures homogènes, ne considère que la redondance logicielle des opérations et pas des communications, et ne peut tolérer qu'une seule faute de processeur. Nous avons appliqué les mêmes hypothèses à notre heuristique dans cette simulation. L'objectif général de cette simulation est de comparer le surcoût dans la longueur de la distribution/ordonnancement introduit par AAA-TP et HBP, en absence et en présence d'une seule défaillance d'un processeur. Enfin, nous avons implémenté ces deux heuristiques AAA-TP et HBP dans l'outil SYNDEX[5].

6.7.1 Les paramètres de simulation

Nous avons appliqué AAA-TP et HBP à un ensemble de graphes d'algorithmes générés aléatoirement et à deux graphes d'architectures : une architecture de $P{=}4$ processeurs et une architecture de $P{=}6$ processeurs. Afin de générer ces graphes aléatoires, nous avons fait varier deux paramètres dans notre générateur de graphes aléatoires[6] :

- le nombre d'opérations : $N = 10, 20, ..., 80$;
- le rapport entre le temps moyen de communication et le temps moyen d'exécution : $CCR = 0.1, 0.5, 1, 2, 5, 10$.

Le surcoût dans la longueur de la distribution/ordonnancement est calculé comme suit :

$$\text{Surcoût} = \frac{\text{longueur(AAA-TP ou HBP)} - \text{longueur(HTR)}}{\text{longueur(HTR)}} * 100$$

où HTR[7] est l'heuristique de distribution/ordonnancement de SYNDEX présentée dans la section 2.6.2 (page 28).

6.7.2 Les résultats

6.7.2.1 Effet de la réplication active sur AAA-TP

Nous avons tracé dans la figure 6.15 la moyenne du surcoût en longueur de la distribution/ordonnancement de 50 graphes aléatoires pour N=50 opérations, $CCR = 0.1, 0.5, 1, 5$ et $\mathcal{N}pf$=1.

Cette figure montre que les performances de AAA-TP augmentent lorsque CCR croît. Ceci s'explique par le fait que l'heuristique AAA-TP est basée sur un ensemble de schémas de transformation (voir figures 6.10 et 6.11) qui permettent de réduire le surcoût en communication lorsque CCR croît. Ici, la réplication en plus de deux copies ($\geq \mathcal{N}pf$+1) de certaines opérations de $\mathcal{A}lg$ peut réduire le temps global de communication [2].

[5] http ://www-rocq.inria.fr/syndex

[6] Notre générateur aléatoire de graphes est présenté dans le chapitre 10.

[7] HTR = Heuristique de distribution/ordonnancement Temps Réel.

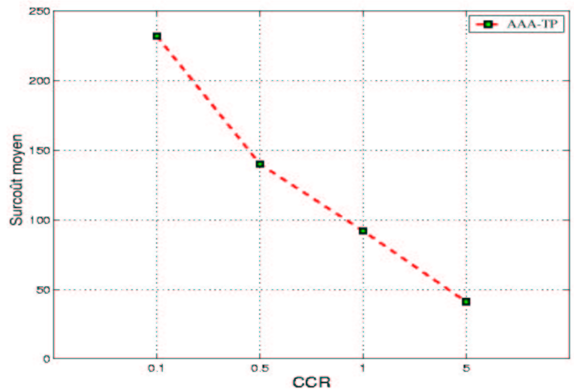


FIG. 6.15 – Effet de la réplication active pour $\mathcal{N}pf = 1$, $P = 6$ et $N = 50$.

6.7.2.2 Comparaison entre AAA-TP et HBP

Nous avons tracé dans la figure 6.16 la moyenne du surcoût en longueur de la distribution/ordonnancement de 50 graphes aléatoires pour CCR=5, P=4, $\mathcal{N}pf$=1, et $N = 10, \ldots, 80$, en absence de défaillance (figure 6.16a) et en présence d'une défaillance d'un seul processeur (figure 6.16b).

La figure 6.16 montre que les performances de AAA-TP et HBP décroissent lorsque le nombre d'opérations croît. Ceci s'explique par le fait que chaque opération est répliquée en deux exemplaires. Dans la plupart des cas, les résultats montrent surtout que AAA-TP est plus efficace que HBP.

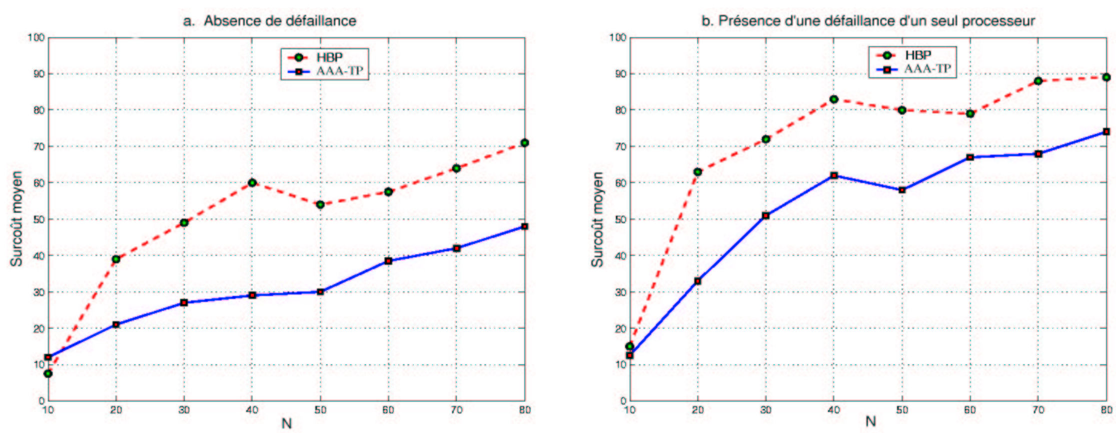


FIG. 6.16 – Effet de N sur AAA-TP et HBP pour $\mathcal{N}pf = 1$, $P = 4$ et $CCR = 5$.

Nous avons tracé dans la figure 6.17 la moyenne du surcoût en longueur de la distribution/ordonnancement de 50 graphes aléatoires pour P=4, N=50, $\mathcal{N}pf$=1, et $CCR = 0.1, 0.5, 1, 2, 5, 10$, en absence de défaillance (figure 6.17a) et en présence d'une défaillance d'un seul processeur (figure 6.17b).

La figure 6.17 montre que les performances de AAA-TP et HBP croissent lorsque CCR croît. Ceci s'explique par la réplication de chaque opération en plus de deux exemplaires afin de réduire les coûts de communications. Cela a pour effet d'augmenter la localité des calculs, ce qui est d'autant plus bénéfique que CCR est grand. En outre, pour $CCR \geq 1$, les résultats montrent que les performances de AAA-TP sont meilleures relativement à HBP.

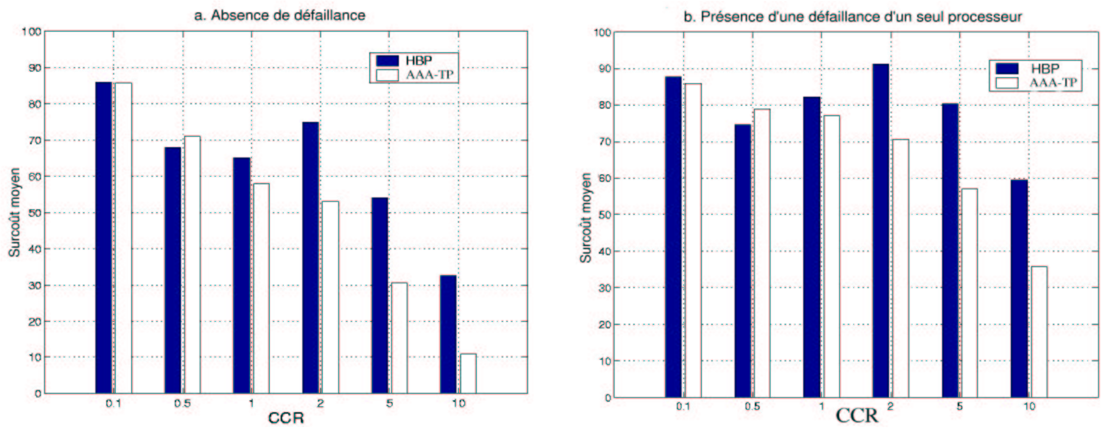


FIG. 6.17 – Effet de CCR sur AAA-TP et HBP pour $\mathcal{N}pf$=1, P=4 et N=50.

6.8 Conclusion

Nous avons proposé dans ce chapitre une méthodologie AAA-TP pour la tolérance aux fautes dans un système distribué réactif embarqué. AAA-TP implante une solution logicielle pour la tolérance aux fautes basée sur la technique de masquage des erreurs. La tolérance aux fautes est obtenue hors-ligne et en deux phases. La première phase s'appuie sur un formalisme de graphes pour transformer une spécification d'un algorithme sans redondance en une spécification avec redondances et relations d'exclusion. La deuxième phase consiste à allouer spatialement et temporellement les composants logiciels de ce nouvel algorithme avec redondances sur les composants matériels de l'architecture. Notre méthodologie AAA-TP est implémentée au sein de l'outil SYNDEX par une heuristique. Les simulations sur des graphes d'algorithmes générés aléatoirement indiquent que cette heuristique a des meilleurs résultats que l'heuristique trouvée dans la littérature ayant les objectifs les plus similaires.

« Tout notre savoir prend ses origines
dans nos perceptions. »
Léonard DE VINCI

Chapitre 7

Méthodologie AAA-TB pour des architectures à liaisons bus

Résumé

Ce chapitre présente une méthodologie pour la génération automatique de distributions/ordonnancements tolérantes aux fautes matérielles. Contrairement au chapitre précédent, la méthodologie que nous présentons ici est adaptée aux architectures matérielles munies d'un réseau de communication composé uniquement de liaisons bus. Elle est basée sur la redondance hybride et la fragmentation des données de communication pour tolérer des fautes temporelles des processeurs et des bus de communication.

7.1 Présentation du problème de tolérance aux fautes

Nous avons présenté, dans le chapitre 6, une méthodologie AAA-TP pour la tolérance aux fautes matérielles adaptée aux architectures à liaisons point-à-point. Elle peut être facilement utilisée aussi pour des architectures à liaisons bus, que nous appelons *architectures multi-bus*. Cependant, ces architectures possèdent une propriété intéressante qui est la connexion de tous ses processeurs à chacun de ses bus de communication. L'exploitation efficace de cette propriété peut réduire considérablement le surcoût en nombre de communications engendrées par AAA-TP. Ainsi, le nombre $\mathcal{N}pf+\mathcal{N}lf+1$ routes disjointes exigé par AAA-TP peut être réduit grâce à cette propriété, ainsi que le nombre de média de communication. De plus, dans le cas point-à-point, la faute d'un des communicateurs d'un lien de communication provoque sa défaillance, ce qui n'est pas le cas pour un bus de communication, d'où la nécessité de modifier le modèle de fautes.

Notre but dans ce chapitre est de résoudre le problème 4, le même que celui résolu par AAA-TP, sauf qu'ici notre solution doit tolérer des fautes des bus de communication au lieu des fautes des liens de communication. Il s'agit de résoudre le problème de la recherche d'une distribution/ordonnancement des composants logiciels du graphe d'algorithme $\mathcal{A}lg$ sur les composants matériels du graphe d'architecture $\mathcal{A}rc$, qui tolére $\mathcal{N}pf$ fautes d'opérateurs de calcul et $\mathcal{N}bf$[1] fautes de bus de communication, tout en minimisant la longueur de cette distribution/ordonnancement afin de satisfaire la contrainte temps réel $\mathcal{R}tc$. Nous dénotons ce nouveau problème par « problème 4^{bus} », et la méthodologie que nous proposons pour la résoudre s'appelle AAA-TB[2]. Afin de bien présenter notre solution, nous présentons tout d'abord le modèle de fautes que nous considérons dans ce chapitre.

7.1.1 Modèle de fautes

Dans notre modèle de fautes, nous utilisons les mêmes hypothèses de défaillances que nous avons définies dans la section 6.1.1, sauf qu'ici :

- nous remplaçons les fautes de liens de communications par des fautes de bus de communication ;
- nous remplaçons aussi le nombre $\mathcal{N}lf$ de fautes de liens de communication par le nombre $\mathcal{N}bf$ de fautes de bus de communication ;
- la défaillance d'un bus de communication peut être *partielle* ou *complète* ; la défaillance complète d'un bus est la conséquence de la défaillance de *tous* ses communicateurs, tandis que la défaillance partielle est la conséquence d'une ou plusieurs défaillances de ses communicateurs à condition qu'*au moins deux* de ses communicateurs restent actifs ; par exemple, sur l'architecture multi-bus de la figure 7.1, la défaillance du bus $B1$ est partielle, tandis que la défaillance du bus B_2 est complète.

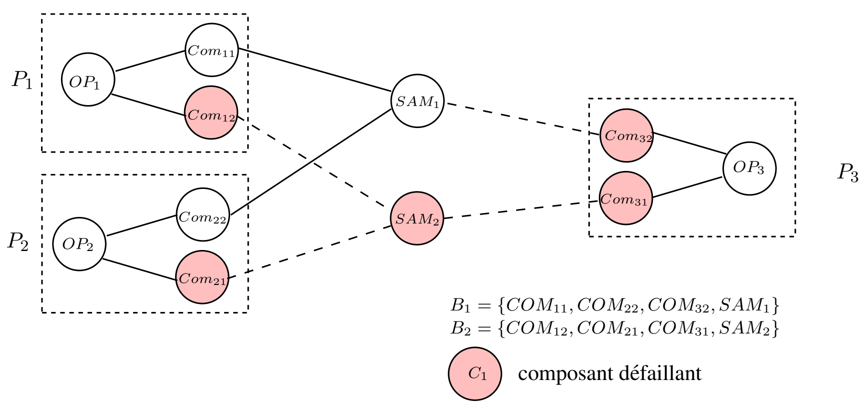


FIG. 7.1 – Défaillance d'un bus de communication

Notre nouvelle hypothèse 4 de défaillance est donc :

Hypothèse 10 *Les fautes matérielles sont des fautes des opérateurs de calculs et des fautes des bus de communication, et la défaillance d'un bus peut être partielle ou complète*

[1] $\mathcal{N}bf$= Number of Bus Failures.
[2] AAA-TB = Adéquation Algorithme Architecture Tolérante aux fautes pour des architectures à liaisons Bus.

Remarque 15 *Puisque la défaillance d'un opérateur de calcul cause la défaillance de son processeur, dans la suite de ce chapitre une faute d'un opérateur de calcul désigne aussi une faute d'un processeur.*

7.2 Principe général de la méthodologie AAA-TB

D'après le théorème 2 (page 78), le problème 4^{bus} de distribution/ordonnan-cement est un problème d'optimisation NP-difficile. Pour résoudre ce problème 4^{bus} en temps polynômial, nous proposons une méthodologie, appelée AAA-TB, qui essaye de trouver une solution proche de la solution optimale. Rappelons que notre objectif n'est pas nécessairement d'avoir une distribution/ordonnancement de longueur minimale mais plutôt qui respecte la contrainte temps réel $\mathcal{R}tc$. À la différence de AAA-TP, la méthodologie AAA-TB implante une solution basée sur la *redondance hybride* et la *fragmentation des données* de communication, comme cela est montré sur la figure 7.2.

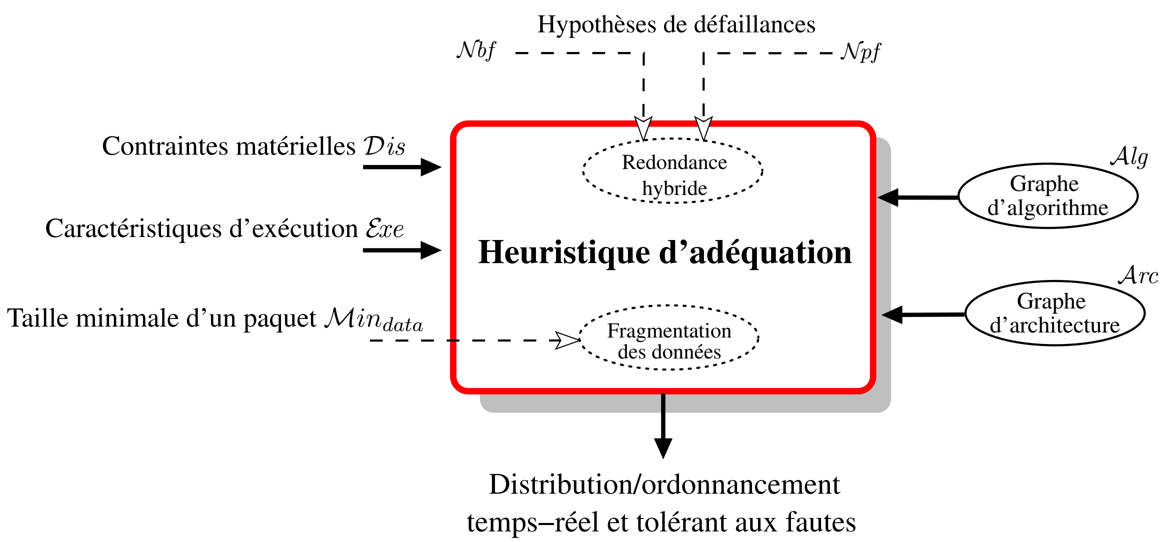


FIG. 7.2 – Méthodologie AAA-TB.

Principe 5 (Redondance hybride) *La redondance hybride des composants logiciels nous semble la plus appropriée pour pouvoir exploiter les deux propriétés de connexion et de défaillance d'un bus de communication.*

L'utilisation de la redondance hybride nécessite un mécanisme spécial de détection et de traitement des erreurs des processeurs et des bus de communication.

Principe 6 (Fragmentation des données) *Afin de recouvrir rapidement les erreurs d'opérateurs de calcul et des bus de communication, AAA-TB utilise une technique de communication basée sur la fragmentation des données de communication en plusieurs paquets de données.*

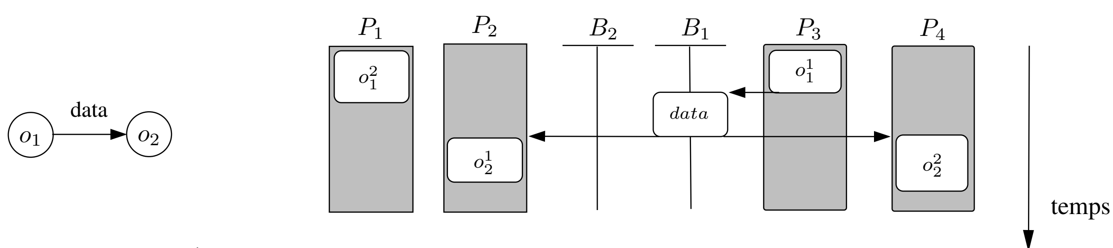
Principe 7 (Tolérance aux fautes arbitraires) *La méthodologie AAA-TB est basée sur une heuristique d'adéquation qui permet de générer hors-ligne une distribution/ordonnancement d'un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$, tolérante à n'importe quelle combinaison arbitraire d'au plus $\mathcal{N}pf$ fautes de processeurs et d'au plus $\mathcal{N}bf$ fautes de bus de communication.*

Afin de bien présenter le principe de la méthodologie pour tolérer les fautes des processeurs et des bus de communication, nous avons choisi de la présenter tout d'abord pour tolérer uniquement les fautes des processeurs, puis pour tolérer uniquement les fautes des bus de communication, et enfin pour tolérer les deux. Et avant tout, nous détaillerons le principe 5 de la redondance hybride et le principe 6 de la fragmentation des données.

7.2.1 Redondance active des opérations et passive des communications

À la différence de la méthodologie AAA-TP, où toutes les opérations (resp. dépendances de données) de $\mathcal{A}lg$ sont répliquées activement sur des processeurs distincts (resp. routes disjointes), AAA-TB est basée sur une technique de redondance hybride : redondance active des opérations et passive des communications. Donc, les opérations de $\mathcal{A}lg$ sont répliquées en plusieurs répliques qui doivent être placées activement sur des processeurs distincts. Quant aux dépendances de données, elles sont répliquées en plusieurs répliques, mais une seule de ces répliques est exécutée, et en cas de défaillance du bus implantant cette réplique ou de son processeur émetteur, une des autres répliques est exécutée, et ainsi de suite.

Par exemple, dans la distribution/ordonnancement de la figure 7.3b, les deux opérations o_1 et o_2, du graphe d'algorithme $\mathcal{A}lg$ de la figure 7.3a, sont répliquées en deux répliques identiques afin de tolérer une faute d'un processeur. Ensuite, une seule réplique o_1^1 de o_1 envoie la dépendance de données $(o_1 \triangleright o_2)$ sur le bus B_1 à toutes les répliques de o_2.



a. Graphe d'algorithme $\mathcal{A}lg$ b. Distribution et ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$

FIG. 7.3 – Redondance active des opérations et passive des communications.

7.2.2 Fragmentation des données de communication

Afin de bien exploiter la redondance matérielle offerte par les architectures multi-bus, la stratégie de communication que nous proposons est basée sur la fragmentation des données de communication en plusieurs paquets de données. La fragmentation de données nous permet de détecter et de recouvrir rapidement les fautes des processeurs et des bus de communication.

Principe 8 *Les données $data$ de chaque dépendance de données ($o_1 \triangleright o_2$) sont fragmentées en k paquets de données : $data^1, \ldots, data^k$, ce qui donne $data = data^1 \bullet \ldots \bullet data^k$. L'opération « $\bullet$ » sert à concaténer deux paquets de données ; elle est associative. Chaque paquet $data_i$ est émis sur le bus B_i. Le nombre k de paquets dépend de trois paramètres : $\mathcal{N}bf$, la taille minimale d'un paquet $\mathcal{M}in_{data}$ et le nombre de bus de communication.*

Par exemple, dans la distribution/ordonnancement de la figure 7.4, l'opération o_1, du graphe d'algorithme $\mathcal{A}lg$ de la figure 7.3a, fragmente les données $data$ de ($o_1 \triangleright o_2$) en deux paquets $data^1$ et $data^2$, et ensuite elle envoie ces deux paquets à l'opération o_2 via les deux bus B_1 et B_2.

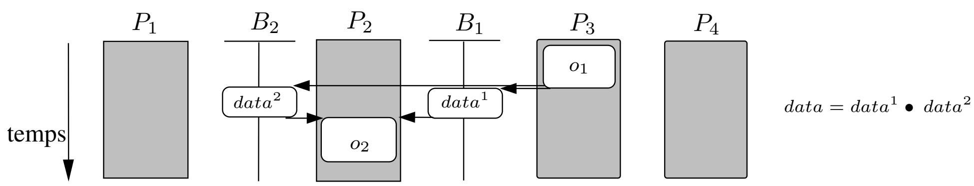


FIG. 7.4 – Fragmentation des données de communications.

7.2.3 Tolérance aux fautes des processeurs

Notre objectif ici est de générer une distribution/ordonnancement, d'un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$, tolérante uniquement aux fautes des processeurs. Pour tolérer $\mathcal{N}pf$ fautes de processeurs, nous répliquons chaque opération o_i de l'algorithme $\mathcal{A}lg$ en $\mathcal{N}pf$+1 répliques exclusives $o_i^1, o_i^2, \ldots, o_i^{\mathcal{N}pf+1}$; l'ensemble de ces répliques est noté $\mathcal{R}ep(o_i)$. Les répliques $\mathcal{R}ep(o_i)$ de o_i sont ordonnées en ordre croissant suivant leur date de fin d'exécution. Puisque nous utilisons la redondance passive des communications, le recouvrement d'erreurs peut augmenter significativement la longueur de la distribution/ordonnancement. Afin d'accélérer le processus de recouvrement des erreurs et donc de minimiser le surcoût en longueur de la distribution/ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$ en présence de défaillances, nous proposons dans AAA-TB une stratégie de communication basée sur la fragmentation des données de communication.

Avant de placer les répliques $\mathcal{R}ep(o_i)$ de l'opération o_i sur $\mathcal{N}pf$+1 processeurs distincts, il nous faut tout d'abord placer leurs dépendances de données ($o_j \triangleright o_i$), pour tous les $o_j \in pred(o_i)$, en utilisant le principe de la fragmentation des données. Donc,

- dans un premier temps, les données $data$ de chaque dépendance ($o_j \triangleright o_i$) de $\mathcal{A}lg$ sont fragmentées en k paquets de données. Le nombre de paquet k est calculé par l'équation suivante :

$$k = \begin{cases} m & \text{si } \frac{T_{data}}{m} \geq \mathcal{M}in_{data} \\ \frac{T_{data}}{\mathcal{M}in_{data}} & \text{sinon} \end{cases} \tag{7.1}$$

où m représente le nombre de bus de communication, T_{data} représente la taille des données de $(o_j \triangleright o_i)$, et $\mathcal{M}in_{data}$ représente la taille minimale d'un paquet de données.

Remarque 16 *Dans le cas où la taille $\frac{T_{data}}{m}$ d'un paquet est supèrieure à $\mathcal{M}ax_{data}$ (la taille maximale d'un paquet de données), chaque paquet $data^i$ est emis sur le bus B_i sous forme de plusieurs sous-paquets de taille infèrieure à $\mathcal{M}ax_{data}$.*

- dans un deuxième temps, *au plus* k' répliques de o_j sont sélectionnées pour envoyer ces k dépendances de données en parallèle à toutes les répliques de o_i, successeur de o_j, via k bus distincts. Puisque chaque réplique de o_j peut envoyer jusqu'à max_k ($0 \leq max_k \leq k$) paquets de $(o_j \triangleright o_i)$, les répliques $\mathcal{R}ep(o_j)$ de o_j peuvent être classées en deux groupes : l'ensemble des répliques primaires, noté $\mathcal{R}ep_{prm}(o_j)$, et l'ensemble des répliques de sauvegarde, noté $\mathcal{R}ep_{svg}(o_j)$. L'ensemble $\mathcal{R}ep_{prm}(o_j)$ contient toutes les répliques de o_j qui ont $max_k > 0$, et l'ensemble $\mathcal{R}ep_{svg}(o_j)$ contient toutes les répliques de o_j qui ont $max_k = 0$. Le rôle des opérations de $\mathcal{R}ep_{prm}(o_j)$ est d'envoyer les données fragmentées de $(o_j \triangleright o_i)$ aux répliques de o_i via k bus distincts. Le rôle des opérations de $\mathcal{R}ep_{svg}(o_j)$ est de surveiller les opérations de $\mathcal{R}ep_{prm}(o_j)$: en cas de défaillance d'un processeur implantant une opération de $\mathcal{R}ep_{prm}(o_j)$, une réplique de $\mathcal{R}ep_{svg}(o_j)$ placée sur un processeur actif sera sélectionnée pour renvoyer les dépendances de données manquantes sur les même bus de communication. Le partitionnement des répliques $\mathcal{R}ep(o_j)$ de o_j en deux groupes $\mathcal{R}ep_{prm}(o_j)$ et $\mathcal{R}ep_{svg}(o_j)$ est réalisé par l'algorithme de la figure 7.6.

Par exemple, dans la figure 7.5, les deux opérations o_1 et o_2 sont répliquées en trois répliques afin de tolérer au plus deux fautes de processeurs. Ensuite, suivant l'équation (7.1), les données $data$ de la dépendance $(o_1 \triangleright o_2)$ sont fragmentées en deux paquets $data^1$ et $data^2$. Ces deux paquets sont envoyés en parallèle par les deux premières répliques o_1^1 et o_1^2 respectivement via B_1 et B_2, et donc $\mathcal{R}ep_{prm}(o_1) = \{o_1^1, o_1^2\}$ et $\mathcal{R}ep_{svg}(o_1) = \{o_1^3\}$. La réplique o_2^1 (resp. o_2^2) étant placée sur le même processeur que o_1^1 (resp. o_1^2), les données $data$ sont reçues via une communication intra-processeur.

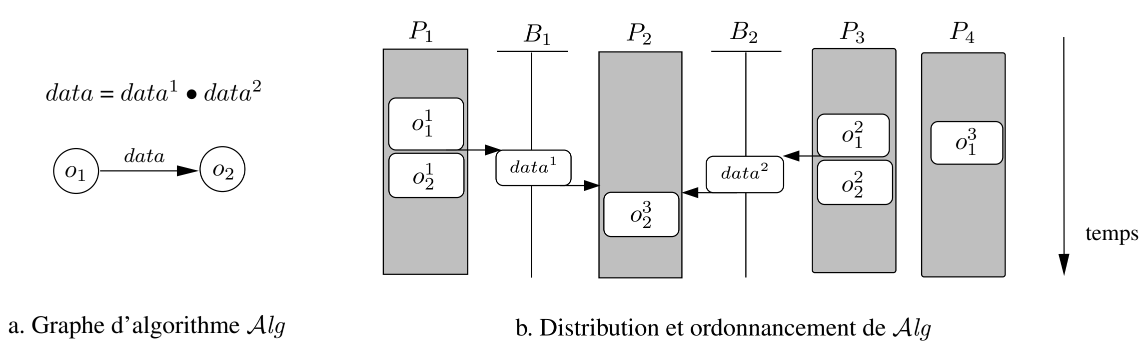


FIG. 7.5 – Tolérance aux fautes des processeurs.

Algorithme fragmentation $(data^1, \ldots, data^k, \mathcal{R}ep(o_j))$
/ $(o_j \triangleright o_i)^l$ désigne les données de $data^l$ */*
/ les opérations de $\mathcal{R}ep(o_j)$ sont ordonnées en ordre croissant suivant leur date de fin d'exécution Et_{exc} */*
Début
$\mathcal{R}ep_{prm}(o_j) := \{o_j^1\}$; */* o_j^1 est la première réplique de $\mathcal{R}ep(o_j)$ */*
$\mathcal{R}ep_{svg}(o_j) := \{o_j^2, \ldots, o_j^{(\mathcal{N}pf+1)}\}$;
Placer/ordonnancer le paquet $(o_j \triangleright o_i)^l$ *sur le bus* B_l*, où* $o_j = o_j^1$;
Soit $St_{com}(o_j \triangleright o_i)^l$ *la date de début d'exécution de* $(o_j \triangleright o_i)^l$;
Soit $\mathcal{R}ep(o_j \triangleright o_i)$ *l'ensemble de toutes les dépendances* $(o_j \triangleright o_i)^l$ *ordonné par* St_{com} *croissante ;*
Pour chaque $(o_j \triangleright o_i)^l$ *de* $\mathcal{R}ep(o_j \triangleright o_i)$ ***faire***
Soit o_j^m *la première opération dans* $\mathcal{R}ep_{svg}$;
Si $Et_{exc}(o_j^m, p_j) \leq St_{com}(o_j \triangleright o_i)^l$
Alors *Re-placer/ordonnancer* $(o_j \triangleright o_i)^l$*, où* $o_j = o_j^m$;
Re-calculer la date de début d'exécution de $(o_j \triangleright o_i)^l$*, où* $o_j = o_j^m$;
$\mathcal{R}ep_{prm}(o_j) := \mathcal{R}ep_{prm}(o_j) \cup o_j^m$;
$\mathcal{R}ep_{svg}(o_j) := \mathcal{R}ep_{svg}(o_j) - o_j^m$;
Sinon *Changer* o_j *par une opération* $o_j^{m'}$ *de* $\mathcal{R}ep_{prm}$;
Re-calculer la date de début d'exécution de $(o_j \triangleright o_i)^l$*, où* $o_j = o_j^{m'}$;
Fin pour chaque
retourner $\mathcal{R}ep_{prm}$ *et* $\mathcal{R}ep_{svg}$;
Fin

FIG. 7.6 – Algorithme de fragmentation de données.

La fragmentation des données présente un avantage très important en présence de défaillances, parce que le recouvrement des erreurs sera *plus rapide* et *moins coûteux* en nombre de communications. Par exemple, si le processeur P_3 implantant la réplique o_1^2 défaille, alors après un certain délai ($timeout$), P_1 détecte l'absence des données $data^2$, et donc la défaillance de P_3 ; ensuite, il renvoie uniquement le paquet $data^2$ sur le même bus B_2, comme cela est illustré sur la figure 7.7.

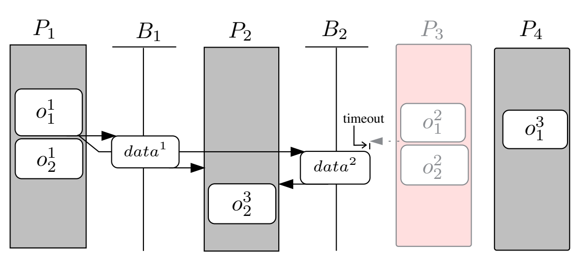


FIG. 7.7 – Recouvrement rapide des erreurs.

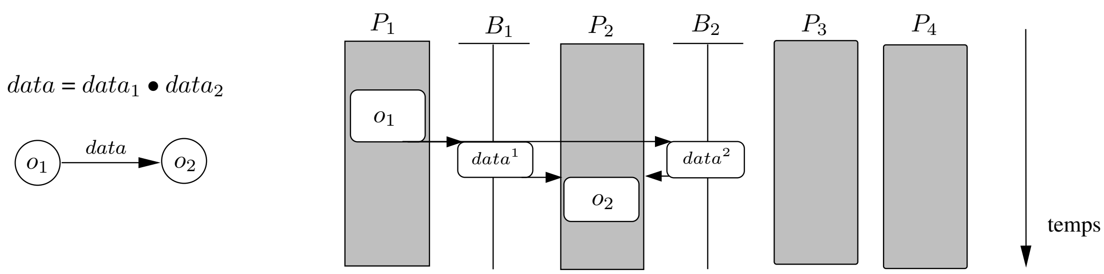
7.2.4 Tolérance aux fautes des bus de communication

Afin de tolérer au plus $\mathcal{N}bf$ fautes de bus de communication, nous utilisons le même principe que nous avons présenté pour tolérer les fautes des processeurs. Sauf qu'ici :

- les opérations ne sont pas répliquées puisque les processeurs sont supposés exempts de faute, et
- chaque dépendance de données $(o_j \triangleright o_i)$ de $\mathcal{A}lg$ est fragmentée en k dépendances de données $(o_j \triangleright o_i)^1, \ldots, (o_j \triangleright o_i)^k$, où :

$$k = \begin{cases} \mathcal{N}bf + 1 & \text{si } \frac{T_{data}}{\mathcal{N}bf+1} \geq \mathcal{M}in_{data} \\ \frac{T_{data}}{\mathcal{M}in_{data}} & \text{sinon} \end{cases} \tag{7.2}$$

- chaque opération o_j envoie, à chaque successeur o_i, chaque dépendance de données $(o_j \triangleright o_i)^k$ via le bus B_k, comme cela est illustré sur la figure 7.8, où $\mathcal{N}bf$=1.



a. Graphe d'algorithme $\mathcal{A}lg$ b. Distribution et ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$

FIG. 7.8 – Tolérance aux fautes des bus de communication.

7.2.5 Tolérance aux fautes des processeurs et des bus de communication

Afin de tolérer $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}bf$ fautes de bus, nous proposons dans cette section une technique basée sur les deux techniques présentées dans la section 7.2.3 (tolérance aux fautes des processeurs) et la section 7.2.4 (tolérance aux fautes des bus). Nous utilisons donc la redondance active des opérations et passive des communications.

La redondance passive et la co-existance des fautes des processeurs et des fautes partielle/complète des bus de communication complique la tâche d'identification des fautes, ce qui peut avoir un effet considérable sur l'augmentation du temps de recouvrement des erreurs. La méthodologie AAA-TB que nous proposons utilise un mécanisme de communication spécial, basé sur la fragmentation des données, qui nous permet :

1. une distinction rapide entre une faute d'un bus et une faute d'un processeur,
2. une distinction rapide entre une faute partielle et une faute complète d'un bus,
3. une identification facile de l'origine d'une faute,

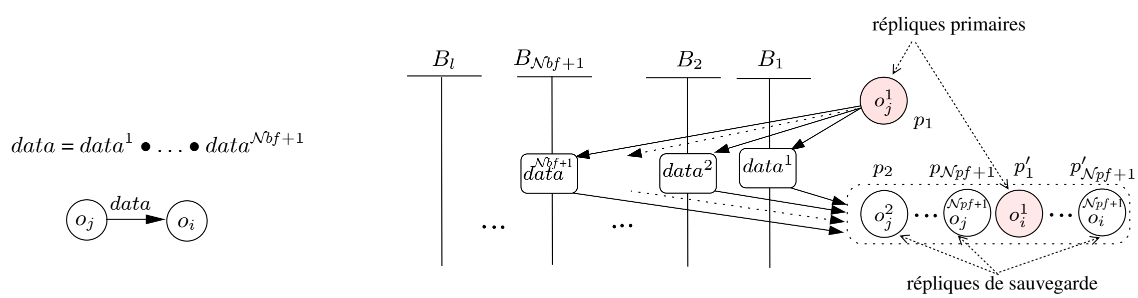
4. une détection rapide des erreurs,
5. un traitement rapide des erreurs.

Dans AAA-TB, nous utilisons les même principes présentés dans les sections 7.2.3 et 7.2.4 pour tolérer $\mathcal{N}pf$ fautes de processeurs et $\mathcal{N}bf$ fautes de bus de communication. Sauf qu'ici, l'ensemble des répliques primaires $\mathcal{R}ep_{prm}(o_j)$ est constitué uniquement de la première réplique o_j^1 de o_j, et l'ensemble des répliques de sauvegarde $\mathcal{R}ep_{svg}(o_j)$ est constitué des autres $\mathcal{N}pf$ répliques restantes de o_j.

Avant de présenter un exemple (figure 7.12) illustrant les cinq points cités précédemment, nous présentons tout d'abord les mécanismes de communication, de détection et de traitement des erreurs.

7.2.5.1 Mécanisme de communication

Supposons que le nombre k, déterminé suivant l'équation (7.2), soit égal à $\mathcal{N}bf$+1. La figure 7.9 illustre la stratégie de communication que nous proposons, où chaque première réplique o_j^1 implantée sur un processeur p_1, appelée réplique primaire, d'une opération o_j fragmente les données $data$ de ($o_j \triangleright o_i$) de ses résultats de sortie en $\mathcal{N}bf$+1 paquets de données ($data^1$, …, $data^{\mathcal{N}bf+1}$), qu'elle envoie en parallèle à toutes les répliques de toutes ses opérations successeurs ($\mathcal{R}ep_{prm}(o_i)$ et $\mathcal{R}ep_{svg}(o_i)$) et à toutes ses propres répliques de sauvegarde ($\mathcal{R}ep_{svg}(o_j)$) via $\mathcal{N}bf$+1 bus distincts.



a. Graphe d'algorithme $\mathcal{A}lg$ b. Redondance passive et fragmentation des données

FIG. 7.9 – Tolérance aux fautes des processeurs et des bus de communication.

Dans le cas où les données $data$ d'une dépendance de données ($o_j \triangleright o_i$) ne peuvent pas être fragmentées suivant l'équation (7.2) en $\mathcal{N}bf$+1 paquets, et afin d'utiliser la même stratégie de communication présentée ci-dessus, nous fragmentons dans un premier temps les données $data$ en k paquets (k<$\mathcal{N}bf$+1) en utilisant l'équation (7.2), d'où $data = data^1 \bullet \ldots \bullet data^k$. Ensuite, dans un deuxième temps nous ajoutons à ces k paquets un ensemble de $\mathcal{N}bf$+1-k messages vides, d'où le nouvel ensemble de $\mathcal{N}pf$+1 paquets $data = data^1 \bullet \ldots \bullet data^k \bullet \emptyset_{k+1} \bullet \emptyset_{\mathcal{N}pf+1}$.

Hypothèse 11 *Afin de bien présenter le mécanisme de détection et de recouvrement d'erreurs, nous supposons pour la dépendance de données ($o_j \triangleright o_i$) que chaque k^{ieme} réplique o_j^k de o_j est placée sur le k^{ieme} processeur p_k de $\mathcal{A}rc$ (figure 7.9).*

7.2.5.2 Détection et traitement des erreurs

Dans le mécanisme de communication de AAA-TB (figure 7.9b), trois cas distinct peuvent se présenter :

1. ***Tous les paquets** $data^m$ **envoyés par** o_j^1 **sont reçus par tous les recepteurs non défaillants*** : dans ce cas, chaque réplique de o_i reconstruit les données $data$ en rassemblant les fragments (les paquets $data^m$) et commence son exécution. Chaque réplique de sauvegarde de $\mathcal{R}ep_{svg}(o_j)$ reçoit aussi tous les paquets $data^m$, qu'elle ignore.
2. ***Aucun des paquets** $data^m$ **envoyés par** o_j^1 **ne sont reçus par tous les recepteurs non défaillants*** : cela représente $\mathcal{N}pf$+1 paquets et que comme par hypothèse le système ne peut pas avoir plus que $\mathcal{N}bf$ fautes de bus, ceci implique forcément la défaillance du processeur p_1 implantant la réplique primaire o_j^1. Donc, tous les processeurs implantant les répliques de o_i et les répliques de sauvegarde $\mathcal{R}ep_{svg}(o_j)$ de o_j déclarent la défaillance du processeur p_1. Ensuite, comme les opérations répliques de o_j sont ordonnées en ordre croissant suivant leur date de fin d'exécution, la première réplique de sauvegarde o_j^2 de $\mathcal{R}ep_{svg}(o_j)$ placée sur un processeur p_2 actif devient la nouvelle réplique primaire ($\mathcal{R}ep_{prm}(o_j) = \{o_j^2\}$), et elle renvoie les mêmes paquets $data^m$ via les mêmes bus utilisés par l'ancienne réplique primaire o_j^1 (voir figure 7.10).

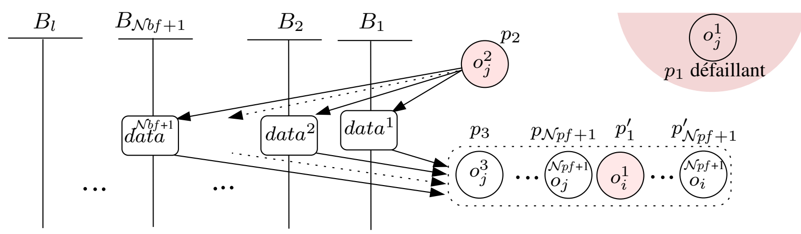


FIG. 7.10 – Tolérance aux fautes des processeurs.

Le même processus de détection et de traitement d'erreurs sera exécuté en cas où les paquets $data^m$ envoyés par la nouvelle réplique primaire o_j^k de $\mathcal{R}ep_{prm}(o_j)$ ne seront pas reçus.

3. ***Certains paquets** $\{data^m, \ldots, data^k\}$ **envoyés par** o_j^1 **ne sont pas reçus par tous les recepteurs non défaillants*** : soient $data^-$ l'ensemble des paquets qui n'ont pas été reçus, et $\mathcal{B}^- = \{B^m, \ldots, B^k\}$ l'ensemble des bus de $\mathcal{A}rc$ qui étaient cencés les transmettre. Dans ce cas, tous les processeurs implantant les répliques de o_i et les répliques de sauvegarde de o_j déclarent la défaillance partielle des bus $\mathcal{B}^-$, et plus particulièrement la défaillance des communicateurs du processeur p_1 reliant les bus $\mathcal{B}^-$. Ensuite, la première réplique de sauvegarde o_j^2 de $\mathcal{R}ep_{svg}(o_j)$ placée sur un processeur p_2 actif devient primaire, et elle renvoie les paquets non reçus via les mêmes bus $\mathcal{B}^-$, et ainsi de suite avec toutes les répliques de

sauvegarde de $\mathcal{R}ep_{svg}(o_j)$ jusqu'à la réception de tous les paquets par toutes les répliques de o_i. Par exemple, sur la figure 7.9b, si p_2 ne reçoit pas les deux paquets $\{data^1, data^2\}$, alors il détecte l'absence des données $\{data^1, data^2\}$ sur les bus $\mathcal{B}^- = \{B_1, B_2\}$, et il envoie ses deux paquets sur les mêmes bus $\mathcal{B}^-$, comme cela est montré sur la figure 7.11, et ainsi de suite avec toutes les répliques de sauvegarde de $\mathcal{R}ep_{svg}(o_j)$ jusqu'à la réception de ces deux paquets.

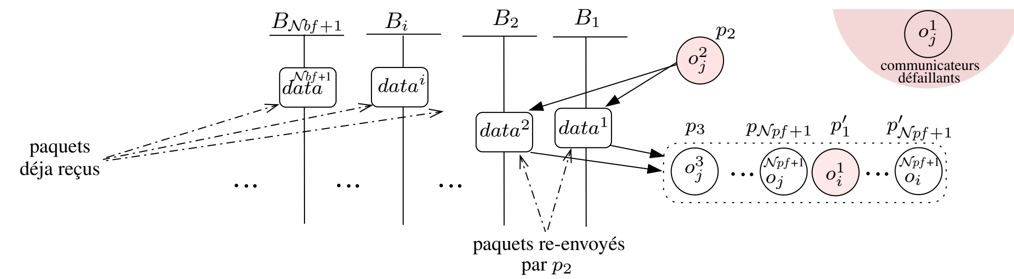


FIG. 7.11 – Tolérance aux fautes des bus de communication.

4. ***Certains paquets*** $\{data^{m'}, \ldots, data^{k'}\}$ ***de*** $\{data^m, \ldots, data^k\}$ ***ne sont pas reçus par tous les recepteurs non défaillants*** : soient $data^{-'}$ l'ensemble des paquets qui n'ont pas été reçus, et $\mathcal{B}^{-'}$ l'ensemble des bus de de $\mathcal{B}^-$ qui étaient cencés les transmettre. Dans ce cas, tous les processeurs actifs implantant les répliques de o_j et de o_i déclarent la défaillance complète de tous les bus $\mathcal{B}^{-'}$. Comme par hypothèse le système ne peut pas avoir plus que $\mathcal{N}bf$ fautes de bus, un des bus B_i non déclaré défaillant ($B_i \notin \mathcal{B}^{-'}$) reste toujours actif. Afin de recouvrir rapidement ces erreurs complètes des bus $\mathcal{B}^{-'}$, nous utilisons le principe de la redondance active. Donc, la réplique primaire de o_j renvoie activement tous les paquets manquants de $data$ via les bus non déclarés défaillants, tandis que les autres répliques de sauvegarde $\mathcal{R}ep_{svg}(o_j)$ surveillent la réplique primaire.

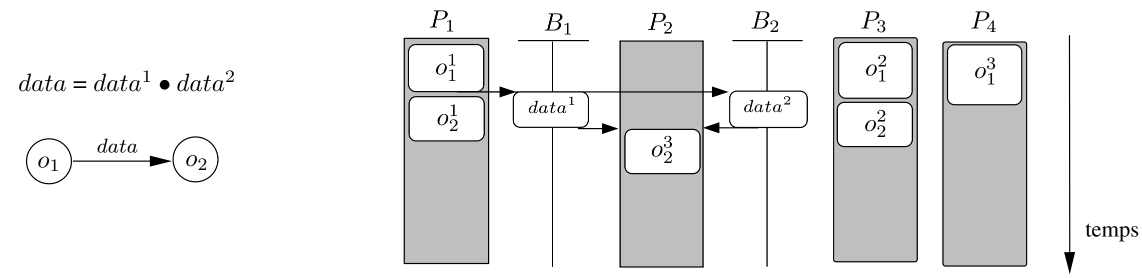
Enfin, le but d'utiliser le paramètre $\mathcal{M}in_{data}$ dans l'équation (7.2) est de contrôler le niveau de la redondance active dans le cas du recouvrement des erreurs complètes des bus $\mathcal{B}^{-'}$ (cas 3). Ce paramètre est lié aussi aux caractéristiques des bus de communication, puisque la taille des données $data$ à envoyer via un bus dépasse souvent la capacité de sa mémoire SAM multi-point. Donc, il est intéressant de fragmenter ces données en plusieurs paquets, et ensuite les envoyer en parallèle via plusieurs bus distincts.

Hypothèse 12 *Le principe de la méthodologie AAA-TB que nous avons présenté dans cette section suppose que les SAM multi-point des bus supportent matériellement la diffusion (broadcast), c'est-à-dire que lors de l'écriture des données de communication $data^m$ de $(o_j \triangleright o_i)^m$ dans une mémoire SAM d'un tel bus, ces données $data^m$ peuvent être lues simultanément par tous les processeurs implantant les répliques de o_i et les répliques de o_j.*

7.2.5.3 Exemple

La distribution/ordonnancement de la figure 7.12b est le résultat de l'application de AAA-TB sur le graphe d'algorithme $\mathcal{A}lg$ de la figure 7.12a et sur un graphe d'architecture $\mathcal{A}rc$, composé de quatre processeurs et de deux bus de communication, pour tolérer au plus deux fautes de processeurs ($\mathcal{N}pf$=2) et au plus une faute d'un bus de communication ($\mathcal{N}bf$=1).

Les deux opérations o_1 et o_2 de $\mathcal{A}lg$ sont répliquées en trois exemplaires afin de tolérer au plus deux fautes de processeurs. Ensuite, suivant l'équation (7.1), les données $data$ de la dépendance $(o_1 \triangleright o_2)$ sont fragmentées en deux paquets $data^1$ et $data^2$. Ces deux paquets sont envoyés en parallèle par la réplique primaire o_1^1 respectivement sur les deux bus de communication B_1 et B_2, et donc $\mathcal{R}ep_{prm}(o_1) = \{o_1^1\}$ et $\mathcal{R}ep_{svg}(o_1) = \{o_1^2, o_1^3\}$. La réplique o_2^1 (resp. o_2^2) étant placée sur le même processeur que o_1^1 (resp. o_1^2), la dépendance de donnée est reçue via une communication intra-processeur.



a. Graphe d'algorithme $\mathcal{A}lg$

b. Distribution et ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$

FIG. 7.12 – Exemple d'application de la méthodologie AAA-TB.

Dans cet exemple :

1. *Distinction rapide entre une défaillance d'un bus et une défaillance d'un processeur* : si la réplique o_2^3 ne reçoit pas les deux paquets $data^1$ et $data^2$, alors c'est un processeur qui est défaillant. En revanche, si la réplique o_2^3 ne reçoit pas un seul des deux paquets, alors c'est un bus qui est défaillant.
2. *Distinction entre une défaillance partielle et une défaillance complète d'un bus* : si la réplique o_2^3 ne reçoit pas un des deux paquets, par exemple $data^2$, alors c'est le bus B_2 qui est partiellement défaillant.
3. *Identification facile de l'origine d'une faute* : si la réplique o_2^3 ne reçoit pas les deux paquets $data^1$ et $data^2$, alors c'est le processeur p_1, implantant o_1^1, qui est défaillant. En revanche, si la réplique o_2^3 ne reçoit pas un des deux paquets, par exemple $data^2$, alors c'est le bus B_2 qui est partiellement défaillant.
4. *Détection rapide des erreurs* : la détection des erreurs des bus de communication se fait juste après la date de fin d'exécution de chaque communication de $data^m$, et la détection des erreurs des processeur se fait juste après la date de fin d'exécution de la dernière communication de tous les paquets de $data$.

5. *Traitement rapide des erreurs* : en cas de défaillance, seuls les paquets manquants seront renvoyés.

7.3 Heuristique de distribution et d'ordonnancement

Les principes de la méthodologie AAA-TB sont implantés par une heuristique de distribution/ordonnancement, qui consiste à mettre en correspondance de manière efficace un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$ pour tolérer *n'importe quelle combinaison arbitraire* d'au plus $\mathcal{N}pf$ fautes de processeurs et d'au plus $\mathcal{N}bf$ fautes de bus de communication.

7.3.1 Présentation de l'heuristique

Notre heuristique de distribution/ordonnancement est un algorithme de construction progressive de type glouton [11]. Nous utilisons, dans cette section, les mêmes notations définies dans la section 6.4.1 (page 91). L'algorithme de l'heuristique se divise en quatre phases : une phase d'initialisation, une phase de sélection, puis une phase de distribution/ordonnancement, et enfin une phase de mise à jour.

ALGORITHME

- ***Entrées =*** *Graphe d'algorithme $\mathcal{A}lg$, graphe d'architecture $\mathcal{A}rc$, $\mathcal{N}pf$, $\mathcal{N}bf$, $\mathcal{M}in_{data}$, caractéristiques d'exécution $\mathcal{E}xe$, contrainte temps réel $\mathcal{R}tc$, contraintes matérielle $\mathcal{D}is$;*
- ***Sortie =*** *Allocation spatiale et temporelle de $\mathcal{A}lg$ sur $\mathcal{A}rc$ tolérante à au plus $\mathcal{N}pf$ fautes de processeurs et à au plus $\mathcal{N}bf$ fautes de bus de communication ;*

INITIALISATION

Initialiser la liste des opérations candidates, et la liste des opérations déjà placées :

$O_{cand}^{(1)} := \{$*opérations de $\mathcal{A}lg$ sans prédécesseurs*$\}$;

$O_{fin}^{(1)} := \emptyset$;

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

Tant que $O_{cand}^{(n)} \neq \emptyset$ ***faire***

SÉLECTION

- *Sélectionner pour chaque opération candidate o_i de $O_{cand}^{(n)}$ un ensemble $\mathcal{P}_{best}$ de $\mathcal{N}pf$+1 processeurs de $\mathcal{A}rc$ qui minimisent la pression d'ordonnancement (équation (2.5)) ;*
- *Sélectionner, parmi les processeurs de $\mathcal{P}_{best}(o_i)$, le meilleur processeur p_{best} qui maximise la pression d'ordonnancement (équation (2.5)) pour chaque opération candidate o_i de $O_{cand}^{(n)}$;*
- *Sélectionner, parmi les couples (o_i, p_{best}), le meilleur couple (o_{best}, p_{best}) qui maximise la pression d'ordonnancement (équation (2.5)) ;*

DISTRIBUTION ET ORDONNANCEMENT

- *Soient $\mathcal{P}_{best}(o_{best})$ les $\mathcal{N}pf$+1 meilleurs processeurs de o_{best} calculés par la phase de « Sélection » ;*
- *Pour chaque $o_j \in pred(o_i)$ fragmenter les données $data$ de la dépendance ($o_j^1 \triangleright o_{best}$) en $\mathcal{N}bf$+1 communications ($data^m$) ;*
- *Placer/ordonnancer toutes les communications ($data^m$) de chaque dépendance sur $\mathcal{N}bf$+1 bus distincts ;*
- *Répliquer la meilleure opération o_{best} en $\mathcal{N}pf$+1 répliques ;*
- *Placer/ordonnancer chaque réplique o_{best}^k sur chaque processeur p_{best}^k de $\mathcal{P}_{best}(o_{best})$.*

MISE À JOUR

- *Mettre à jour la liste des opérations candidates et déjà placées :*
  $O_{fin}^{(n+1)} := O_{fin}^{(n)} \cup \{o_{best}\}$;
  $O_{cand}^{(n+1)} := O_{cand}^{(n)} - \{o_{best}\} \cup \left\{o_{new} \in succ(o_{best}) \mid pred(o_{new}) \subseteq O_{fin}^{(n+1)}\right\}$;

Fin tant que

FIN DE L'ALGORITHME

7.3.1.1 Phase d'initialisation

Cette phase consiste à initialiser la liste des opérations candidates $O_{cand}^{(1)}$ avec les opérations sans prédécesseur. Donc, les seules opérations implantables à cette première étape de l'heuristique sont les opérations d'entrées (capteurs). De plus, la liste des opérations déjà placées $O_{fin}^{(1)}$ est vide.

7.3.1.2 Phase de sélection

Dans cette phase, la candidate la plus urgente o_{best}^k est sélectionnée parmi toutes les opérations candidates pour être placée et ordonnancée. La règle de sélection choisie repose sur la pression d'ordonnancement donnée par l'équation (2.5). Donc, la meilleure candidate o_{best} qui maximise cette fonction est sélectionnée, ainsi que ses $\mathcal{N}pf$+1 meilleurs processeurs $\mathcal{P}_{best}$.

7.3.1.3 Phase de distribution/ordonnancement

Après avoir sélectionné la meilleure candidate o_{best}, cette phase consiste en un premier temps à répliquer cette candidate en $\mathcal{N}pf$+1 répliques, et en un deuxième temps, à placer/ordonnancer chaque réplique o_{best}^k de o_{best} sur le k^{ieme} processeur p_{best}^k de $\mathcal{P}_{best}$. Avant de placer/ordonnancer ces répliques, chaque dépendance de données ($o_j \triangleright o_{best}$) sera fragmentée en $\mathcal{N}bf$+1 paquets de communication qui seront placés/ordonnancés sur $\mathcal{N}bf$+1 bus distincts.

7.3.1.4 Phase de mise à jour

Cette phase consiste à mettre à jour tout d'abord la liste des opérations déjà placées et ensuite la liste des opérations candidates. L'opération o_{best} sera supprimée de la liste des candidates $O_{cand}^{(n)}$, et les nouvelles opérations ajoutées à cette liste sont les opérations de $\mathcal{A}lg$ qui ont toutes leurs prédécesseurs dans la liste des opérations déjà placées $O_{fin}^{(n+1)}$.

7.4 Simulations

Afin d'évaluer l'heuristique de distribution/ordonnancement de AAA-TB, nous l'avons comparée avec l'heuristique HTSF[3] proposée par Girault et al. dans [33] ; cette dernière tolère uniquement les fautes des processeurs, pour des architectures avec bus ; pour cela elle utilise la réplication active des opérations et la réplication passive des communications, et donc elle ne fragmente pas les données ; elle prend en paramètre le nombre de fautes de processeurs à tolérer. AAA-TB prend en paramètre le nombre de fautes de processeurs et le nombre de fautes de bus à tolérer. Ainsi, AAA-TB$(\mathcal{N}pf, 0)$ est exactement la même heuristique que HTSF$(\mathcal{N}pf)$.

L'objectif général de nos simulations est de comparer le surcoût dans la longueur de la distribution/ordonnancement introduit par AAA-TB et HTSF, et ceci en absence de défaillance. Nous avons implémenté ces deux heuristiques AAA-TB et HTSF dans l'outil SynDEx.

7.4.1 Les paramètres de simulation

Nous avons appliqué AAA-TB et HTSF à un ensemble de graphes d'algorithmes générés aléatoirement et un graphe d'architecture composé de cinq processeurs et de quatre bus de communication. Afin de générer ces graphes aléatoires, nous avons fait varier deux paramètres dans notre générateur aléatoire de graphes (présenté dans le chapitre 10) :

- le nombre d'opérations : $N = 20, 40, 60, 80, 100$;
- le rapport entre le temps moyen de communication et le temps moyen d'exécution : $CCR = 0.1, 0.5, 1, 5, 10$.

Le surcoût dans la longueur de la distribution/ordonnancement est calculé comme suit :

$$\text{Surcoût} = \text{longueur}(\text{HTSF}(\mathcal{N}pf)) - \text{longueur}(\text{AAA-TB}(\mathcal{N}pf, \mathcal{N}bf))$$

7.4.2 Les résultats

Afin d'évaluer l'effet du nombre d'opération N et de $\mathcal{N}bf$ sur notre heuristique, nous avons tracé dans la figure 7.13 la moyenne du surcoût en longueur de la distribution/ordonnancement de 50 graphes aléatoires pour $N = 20, \ldots, 100$, P=5, $\mathcal{N}pf$=0, CCR=1 et $\mathcal{N}bf = 1, 2, 3$, et ceci en absence de défaillance. Cette figure montre deux choses :

- Tout d'abord les performances de AAA-TB augmentent, relativement à HTSF, lorsque $\mathcal{N}bf$ croît. Ce résultat, apparemment paradoxal, s'explique par le fait que AAA-TB utilise la fragmentation de données, ce qui réduit grandement le surcoût en communication dans le cas d'une

[3] HTSF = Heuristique de Tolérance aux fautes Sans Fragmentation de données.

architecture multi-bus. Au contraire, HTSF ne fragmente pas les données et donc utilise très mal les bus de l'architecture.

- De plus, plus $\mathcal{N}bf$ est grand, meilleures sont les performances de AAA-TB. La raison en est que le taux de fragmentation des données dans AAA-TB est déterminé en supposant que le nombre de bus disponibles dans l'architecture est égal à $\mathcal{N}bf$+1. Autrement dit, AAA-TB(0,1) ne fragmente pas les données autant qu'elle le pourrait sur une architecture à 4 bus. Ce résultat indique donc qu'il faudrait améliorer le taux de fragmentation des données dans AAA-TB afin de mieux exploiter les bus de l'architecture.

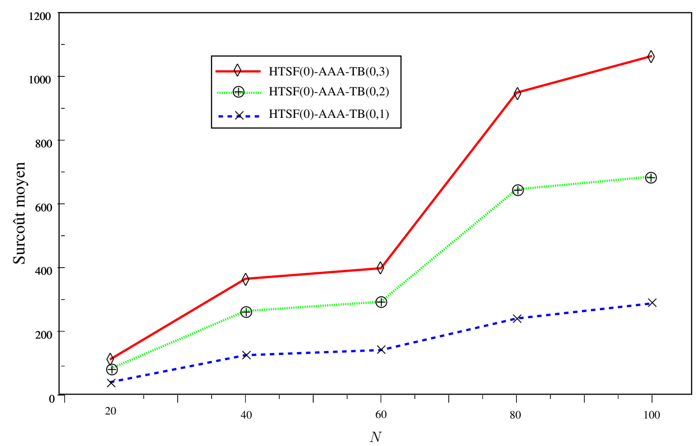


FIG. 7.13 – Effet de N et $\mathcal{N}bf$ pour $\mathcal{N}pf = 0$, $P = 5$ et $CCR = 1$.

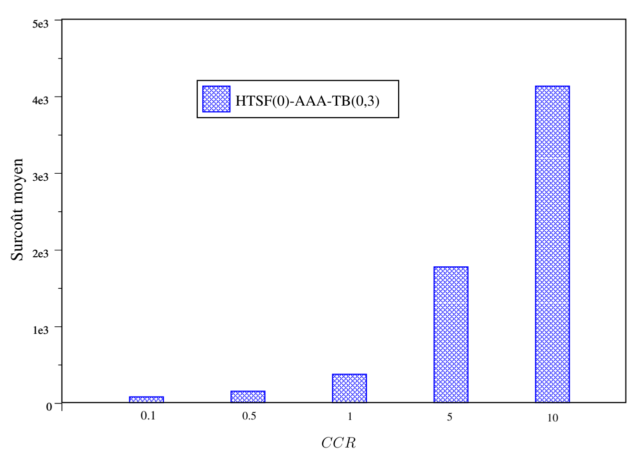


FIG. 7.14 – Effet de CCR pour $\mathcal{N}pf = 1$, $\mathcal{N}bf = 2$, $P = 5$ et $N = 40$.

Afin d'évaluer l'effet de CCR sur notre heuristique, nous avons tracé dans la figure 7.14 la moyenne du surcoût en longueur de la distribution/ordonnancement de 50 graphes aléatoires pour P=5, N=40, $\mathcal{N}pf$=1, $\mathcal{N}bf$=2 et $CCR = 0.1, 0.5, 1, 5, 10$, en absence de défaillance. Cette figure montre aussi que les performances de AAA-TB sont meilleures relativement à HTSF.

7.5 Conclusion

Nous avons proposé dans ce chapitre une méthodologie AAA-TB de tolérance aux fautes pour des systèmes distribués réactifs embarqués. AAA-TB implante une solution logicielle pour la tolérance aux fautes basée sur la redondance hybride : redondance active des opérations et passive des communications. L'utilisation de la redondance hybride nécessite un mécanisme spécial de détection et de traitement des erreurs des processeurs et des bus de communication. Dans le but de réduire la latence (délai entre la production et la détection de l'erreur), AAA-TB utilise la fragmentation des données qui permet : une distinction rapide entre une faute d'un bus et une faute d'un processeur, une distinction rapide entre une faute partielle et une faute complète d'un bus, une identification facile de l'origine d'une faute, une détection rapide des erreurs, et enfin un traitement rapide des erreurs.

« Les gens sont en général plus convaincus par les évidences qu'ils découvrent eux-même que par celles des autres »
Blaise PASCAL

Chapitre 8

Méthodologie AAA-F de génération d'ordonnancements distribués temps réel et fiables

Résumé

Ce chapitre présente une méthodologie originale, basée sur la théorie d'ordonnancement et la transformation de graphe, pour la conception des systèmes réactifs fiables. Elle consiste à générer une distribution/ordonnancement d'un algorithme sur une architecture qui optimise deux critères antagonistes, qui sont la minimisation de la durée d'exécution de l'algorithme sur l'architecture et la maximisation de la fiabilité de cette distribution/ordonnancement. La transformation de graphe permet d'approximer le calcul de la fiabilité de cette distribution/ordonnancement de façon à rendre sa complexité polynomiale.

8.1 Présentation du problème bi-critères de temps réel et de fiabilité

Rappelons que notre travail s'inscrit dans l'objectif global de la conception des systèmes réactifs sûrs de fonctionnement. Nous avons présenté dans les deux chapitres précédents 6 et 7 deux méthodologies pour la conception de ces systèmes, où la sûreté de fonctionnement est obtenue par la tolérance aux fautes. Dans ce chapitre, nous nous intéressons à une mesure qui permet d'évaluer la sûreté de fonctionnement d'un système, qui est la fiabilité (cf. section 3.3, page 40). À la différence des deux méthodologies AAA-TP et AAA-TB, l'objectif de ce chapitre est de proposer une solution à un problème bi-critères de génération automatique de distributions/ordonnancements temps réel et fiables. Le premier critère consiste à satisfaire une contrainte temps réel $\mathcal{R}tc_{obj}$, et le deuxième critère consiste à satisfaire une contrainte de fiabilité $\mathcal{F}_{obj}$.

Remarque 17 *Nous utilisons dans ce chapitre les mêmes modèles d'algorithme et d'exécution que ceux présentés respectivement dans la section 4.2 (page 48) et la section 4.4.1 (page 58), et le modèle classique de l'architecture matérielle présenté dans la section 4.3 (page 52).*

Hypothèse 13 *Pour des raisons de lisibilité, nous supposons que l'architecture est complètement connectée. La solution que nous proposons dans ce chapitre peut être facilement étendue pour des architectures non complètement connectées.*

Afin de bien présenter ce problème bi-critères, nous présentons tout d'abord le modèle de fautes.

8.1.1 Modèle de fautes

Dans notre modèle de fautes, nous supposons, en plus de l'hypothèse 1 (cf. section 1, page 39) et les hypothèses 2, 3, 4, 7, 8 (cf. section 8, page 76), les deux hypothèses suivantes :

Hypothèse 14 *La défaillance du processeur p_i suit une loi de Poisson à paramètre constant λ_i, appelé taux de défaillance.*

Hypothèse 15 *La défaillance du lien de communication $l_{i,j}$ suit une loi de Poisson à paramètre constant $\mu_{i,j}$, appelé taux de défaillance.*

8.1.2 Données du problème

Le but de ce chapitre est de résoudre le problème de la recherche d'une distribution/ordonnancement des composants logiciels du graphe d'algorithme sur les composants matériels du graphe d'architecture, tout en *minimisant sa longueur* $\mathcal{R}tc_{cal}$ afin de satisfaire un critère temps réel $\mathcal{R}tc_{obj}$, et en *maximisant* sa fiabilité $\mathcal{F}_{cal}$ afin de satisfaire un critère de fiabilité $\mathcal{F}_{obj}$. Ce problème bi-critères a été abordé d'une façon générale dans la section 3.3.3. Il peut être formalisé comme suit :

Problème 5 *Étant donnés :*

- *une architecture matérielle hétérogène* $\mathcal{A}rc$ *composée d'un ensemble* P *de processeurs et d'un ensemble* L *de liens de communication section 4.3 (page 52) :*

$$P = \{\ldots, p_i, \ldots, p_j, \ldots\}, L = \{\ldots, l_{i,j}, \ldots\}$$

- *un algorithme* $\mathcal{A}lg$ *composé d'un ensemble* E *de dépendances de données et d'un ensemble* O *d'opérations (cf. section 4.2, page 48) :*

$$O = \{\ldots, o_i, \ldots, o_j, \ldots\}, E = \{\ldots, (o_i \triangleright o_j), \ldots\}$$

- *des caractéristiques d'exécution* $\mathcal{E}xe$ *des composants logiciels de* $\mathcal{A}lg$ *sur les composants matériels de* $\mathcal{A}rc$ *(cf. section 4.4.1, page 58),*
- *un ensemble de contraintes matérielles* $\mathcal{D}is$ *(cf. section 4.4.2, page 60),*
- *les taux de défaillances* $\mathcal{T}_{def}$ *des processeurs (exprimés par* λ_i *pour chaque processeur* p_i*), et des liens de communication (exprimés par* $\mu_{i,j}$ *pour chaque lien* $l_{i,j}$*) :*

$$\mathcal{T}_{def} = \{\ldots, \lambda_i, \ldots\} \cup \{\ldots, \mu_{i,j}, \ldots\}$$

- *une fonction de calcul de la fiabilité de la distribution/ordonnancement* $\mathcal{F}_{cal}$*,*
- *un objectif de fiabilité* $\mathcal{F}_{obj}$*,*
- *une fonction de calcul de la longueur de la distribution/ordonnancement* $\mathcal{R}tc_{cal}$*,*
- *un objectif de temps réel* $\mathcal{R}tc_{obj}$*,*

il s'agit de trouver une application $\mathcal{A}$ *qui projette chaque opération (resp. dépendance de données) de* $\mathcal{A}lg$ *sur un processeur (resp. lien de communication) de* $\mathcal{A}rc$*, et qui lui assigne un ordre d'exécution* t_k *sur son processeur (resp. lien) :*

$$\begin{array}{rccl} \mathcal{A}: & \mathcal{A}lg & \longrightarrow & \mathcal{A}rc \\ & \mathcal{A}lg_i & \longmapsto & \mathcal{A}(\mathcal{A}lg_i) = (\mathcal{A}rc_j, t_k) \end{array}$$

qui respecte $\mathcal{D}is$*, qui minimise* $\mathcal{R}tc_{cal}$*, qui maximise* $\mathcal{F}_{cal}$*, et qui satisfait les deux critères de fiabilité* $\mathcal{F}_{obj}$ *et de temps réel* $\mathcal{R}tc_{obj}$*, c'est-à-dire que* $\mathcal{R}tc_{cal}$ *doit être inférieure ou égale à* $\mathcal{R}tc_{obj}$*, et* $\mathcal{F}_{cal}$ *doit être supérieure ou égale à* $\mathcal{F}_{obj}$*.*

8.2 Principe général de la méthodologie AAA-F

Le problème 5 est un problème d'optimisation NP-difficile, puisque il s'agit ici d'optimiser un critère temps réel et un critère de fiabilité.

Théorème 3 *Soient un algorithme constitué de plusieurs composants logiciels dépendants, et une architecture matérielle hétérogène constituée de plusieurs processeurs. La distribution/ordonnancement d'un tel algorithme sur une telle architecture visant la minimisation de la longueur et la maximisation de la fiabilité de la distribution/ordonnancement est un problème NP-difficile.*

Preuve 1 *Selon le théoreme 1 (cf. section 1, page 78), le problème 5 sans l'objectif de maximisation de fiabilité est un problème NP-difficile. Donc, si on considère un nouvel objectif d'optimisation de la fiabilité, ce problème 5 reste toujours NP-difficile.* □

Pour résoudre ce problème en temps polynômial, nous proposons une méthodologie originale, appelée AAA-F[1], qui essaye de trouver une solution valide la plus proche possible de la solution optimale. Cette solution doit vérifier les deux critères de temps réel et de fiabilité avant la mise en exploitation du système.

Principe 9 *Étant donné que la redondance est l'un des moyens les plus utilisées dans la littérature pour augmenter la fiabilité d'un système, et puisque nous visons des systèmes embarqués à défaillances temporelles, AAA-F implante une solution logicielle (principe 1, section 6.2) basée sur la redondance active des composants logiciels. Cela nous parait la solution la plus appropriée pour atteindre les deux exigences d'embarquabilité et de fiabilité de ces systèmes.*

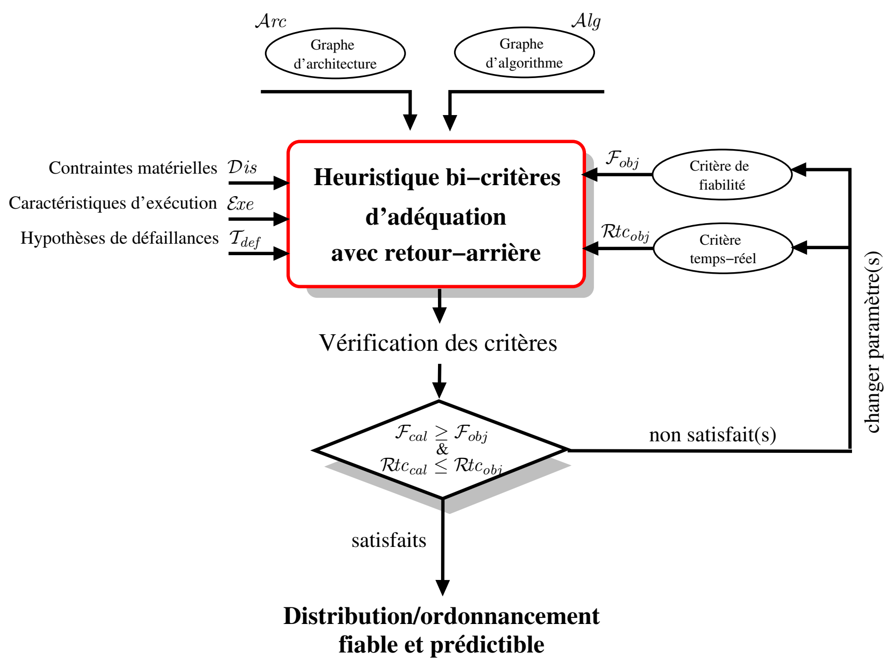


FIG. 8.1 – Méthodologie AAA-F.

La méthodologie AAA-F est basée sur une heuristique d'adéquation qui consiste à mettre en correspondance de manière efficace le graphe d'algorithme $\mathcal{A}lg$ sur le graphe d'architecture $\mathcal{A}rc$, tout en respectant les deux critères $\mathcal{R}tc_{obj}$ et $\mathcal{F}_{obj}$, comme cela est montré sur la figure 8.1. Cette heuristique d'adéquation est un algorithme glouton à construction progressive, de type *retour-arrière*, c'est-à-dire qu'elle possède plusieurs points de reprise qui sont utilisés au cas où, à l'étape (n)

[1]AAA-F : Adéquation Algorithme Architecture Fiable.

de la distribution/ordonnancement, l'heuristique échoue à satisfaire un des deux critères $\mathcal{R}tc_{obj}$ et $\mathcal{F}_{obj}$.

Avant de présenter l'heuristique bi-critères de AAA-F, nous présentons tout d'abord les deux fonctions qui permettent de calculer la longueur $\mathcal{R}tc_{cal}$ et la fiabilité $\mathcal{F}_{cal}$ d'une telle distribution/ordonnancement avec redondance logicielle.

Remarque 18 *Nous utilisons dans la suite de ce chapitre les mêmes notations définies dans la section 6.4.1 (page 91).*

8.2.1 Calcul de la longueur d'une distribution/ordonnancement

Étant donné que la méthodologie AAA-F utilise une redondance logicielle de type « active », il est possible de calculer la longueur $\mathcal{R}tc_{cal}$ d'une telle distribution/ordonnancement avec redondance logicielle par l'équation suivante :

$$\mathcal{R}tc_{cal} = \max_{p_j \in \mathcal{A}rc} \max_{o_i \in \mathcal{O}p(p_j)} Et_{exc}(o_i, p_j) \tag{8.1}$$

8.2.2 Calcul de la fiabilité d'une distribution/ordonnancement

Il existe dans la littérature plusieurs méthodes, basées sur la théorie d'ordonnancement, pour calculer la fiabilité d'une telle distribution/ordonnancement. Elles diffèrent selon plusieurs critères, tels que les moyens d'obtention de ce calcul, le type de mesure de fiabilité, et les caractéristiques du système (système orienté mission, système à exécution continue, système à maintenance, etc). Par exemple, certaines méthodes définissent la fiabilité par la probabilité de bon fonctionnement d'un système durant sa mission [73, 64]. Dans AAA-F, nous utilisons la même définition de fiabilité que [6, 40], qui est la plus adaptée à notre modèle d'exécution de l'algorithme sur l'architecture. Donc, la fiabilité est définie par « *la probabilité qu'une opération (resp. dépendance de données) est exécutée correctement sur son processeur (resp. lien de communication) durant un cycle d'exécution du graphe d'algorithme sur le graphe d'architecture* ».

Cependant, le calcul de la fiabilité d'une telle distribution/ordonnancement, tel qu'il est présenté dans [6, 40], n'est pas adapté aux distributions/ordonnancements avec redondance. Nous présentons successivement le calcul de la fiabilité pour une distribution/ordonnancement sans et avec redondance logicielle. Dans ce travail, nous visons des systèmes réactifs non réparables, c'est-à-dire que les composants matériels défaillants ne seront pas remis en service ou échangés pendant l'exploitation du système. Donc, nous avons décidé d'utiliser les *diagrammes de fiabilité* [14] pour tous nos calculs de fiabilité.

8.2.2.1 Distribution/ordonnancement sans redondance logicielle

Dans ce cas, un tel système peut être représenté par un ensemble de composants en série [14] :

Définition 43 *Deux éléments sont en série si le fonctionnement des deux est nécessaire pour assurer le fonctionnement de l'ensemble.*

Considérons par exemple le système de la figure 8.2. Il est constitué :

- d'une architecture $\mathcal{A}rc$ (figure 8.2b) composée de deux processeur p_1 et p_2, de taux de défaillances respectifs $\lambda_1 = 0.00001$ et $\lambda_2 = 0.0001$; ces deux processeurs sont reliés par un lien de communication $l_{1,2}$ de taux de défaillances $\mu_{1,2} = 0.001$,
- d'un algorithme $\mathcal{A}lg$ (figure 8.2a) composé de deux opérations o_1 et o_2, et d'une dépendance de données $(o_1 \rhd o_2)$, les durées d'exécutions $\mathcal{E}xe$ sont : $\mathcal{E}xe(o_1, p_1) = 1$, $\mathcal{E}xe(o_1, p_2) = 2$, $\mathcal{E}xe(o_2, p_1) = 1$, $\mathcal{E}xe(o_2, p_2) = 2$ et $\mathcal{E}xe(o_1 \rhd o_2, l_{1,2}) = 1$.

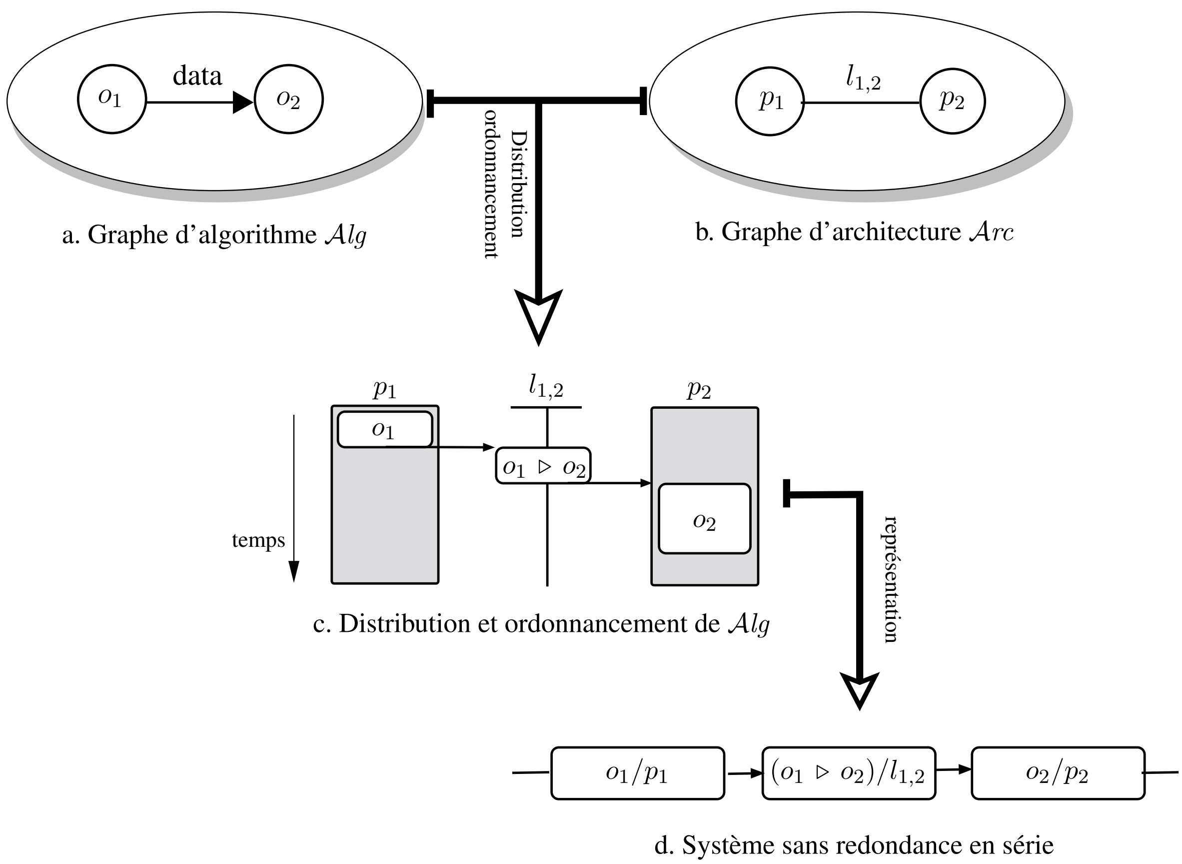


FIG. 8.2 – Exemple d'un système sans redondance en série.

Supposons que o_1 (resp. o_2) est placée sur le processeur p_1 (resp. p_2), et ainsi la dépendance de données $(o_1 \rhd o_2)$ est placée sur le lien $l_{1,2}$, comme cela est montré sur la figure 8.2c. Dans ce cas, le système peut être vu comme un système constitué de trois composants « o_1/p_1 », « $(o_1 \rhd o_2)/l_{1_2}$ » et « o_2/p_2 » en série (figure 8.2d).

La probabilité qu'un système en série fonctionne correctement est définie par la probabilité que *tous* ses composants fonctionnent correctement. Elle est donnée par la formule suivante [40] :

$$\mathcal{F}_{cal}^{serie} = \prod_{o_i \in \mathcal{A}lg} \mathcal{F}_{cal}(o_i) \prod_{(o_i \triangleright o_j) \in \mathcal{A}lg} \mathcal{F}_{cal}(o_i \triangleright o_j)$$

Si l'opération o_i est placée sur le processeur p_k de taux de défaillance λ_k, et si la dépendance de données $(o_i \triangleright o_j)$ est placée sur le lien $l_{k,k'}$ de taux de défaillance $\mu_{k,k'}$, alors :

$$\begin{aligned} \mathcal{F}_{cal}(o_i) &= e^{-\lambda_k \; \mathcal{E}xe(o_i,p_k)} \\ \mathcal{F}_{cal}(o_i \triangleright o_j) &= e^{-\mu_{k,k'} \; \mathcal{E}xe(o_i \triangleright o_j, l_{k,k'})} \end{aligned} \tag{8.2}$$

donc,

$$\begin{aligned} \mathcal{F}_{cal}^{serie} &= \prod_{o_i \in \mathcal{A}lg} e^{-\lambda_k \; \mathcal{E}xe(o_i,p_k)} \prod_{(o_i \triangleright o_j) \in \mathcal{A}lg} e^{-\mu_{k,k'} \; \mathcal{E}xe(o_i \triangleright o_j, l_{k,k'})} \\ &= e^{-\sum_{o_i \in \mathcal{A}lg} \lambda_k \; \mathcal{E}xe(o_i,p_k)} \quad e^{-\sum_{(o_i \triangleright o_j) \in \mathcal{A}lg} \mu_{k,k'} \; \mathcal{E}xe(o_i \triangleright o_j, l_{k,k'})} \\ &= e^{-\left(\sum_{o_i \in \mathcal{A}lg} \lambda_k \; \mathcal{E}xe(o_i,p_k) + \sum_{(o_i \triangleright o_j) \in \mathcal{A}lg} \mu_{k,k'} \; \mathcal{E}xe(o_i \triangleright o_j, l_{k,k'}) \right)} \end{aligned} \tag{8.3}$$

Par exemple, la fiabilité $\mathcal{F}_{cal}^{serie}$ du système en série de la figure 8.2d est :

$$\mathcal{F}_{cal}^{serie} = e^{-\left(\lambda_1 \mathcal{E}xe(o_1,p_1) + \lambda_2 \mathcal{E}xe(o_2,p_2) + \mu_{1,2} \mathcal{E}xe(o_1 \triangleright o_2, l_{2,2})\right)} = e^{-\left((0.00001\ 1 + 0.0001\ 2) + (0.001\ 1)\right)} = 0.99879$$

Sa durée d'exécution totale est calculée par l'équation (8.1) comme suit :

$$\mathcal{R}tc_{cal} = \max\ \left(Et_{exc}(o_1,p_1)\ ,\ Et_{exc}(o_2,p_2)\right) \ = \ \max\ \left(1\ ,\ 4\right) = 4$$

8.2.2.2 Distribution/ordonnancement avec redondance logicielle

L'utilisation de la redondance logicielle donne lieu à trois types de systèmes : système en parallèle, système en série/parallèle et système en structure quelconque.

Système parallèle

Dans le cas où un système avec redondance logicielle est constitué uniquement de composants en parallèle, il est appelé *système en parallèle* [14] :

Définition 44 *Deux éléments sont en parallèle si le fonctionnement d'au moins un des deux est suffisant pour assurer le fonctionnement de l'ensemble.*

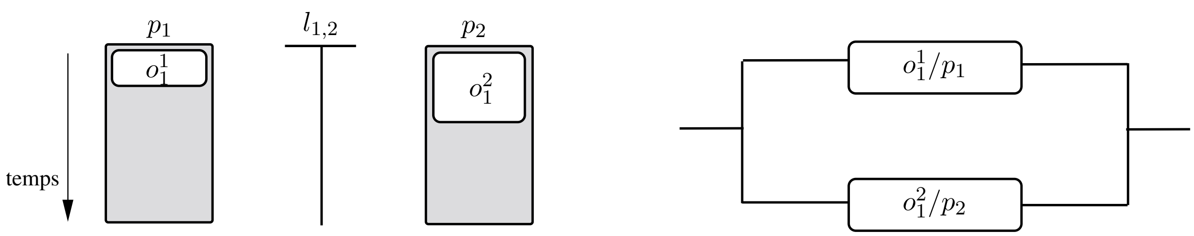
La probabilité qu'un système en parallèle fonctionne correctement est définie par la probabilité qu'au moins *un* de ses composants fonctionne correctement. Elle est donnée par la formule suivante [56] :

$$\mathcal{F}_{cal}^{parallele} = 1 - \prod_{o_i \in \mathcal{A}lg} \big(1 - \mathcal{F}_{cal}(o_i)\big)$$

Suivant l'équation 8.2, on obtient alors :

$$\mathcal{F}_{cal}^{parallele} = 1 - \prod_{o_i \in \mathcal{A}lg} \big(1 - e^{-\lambda_k \ \mathcal{E}xe(o_i,p_k)}\big) \tag{8.4}$$

Considérons un graphe $\mathcal{A}lg$ composé d'une seule opération o_1 et le graphe $\mathcal{A}rc$ de la figure 8.2b) de la figure . Supposons que o_1 soit répliquée en deux copies : une copie o_1^1 placée sur le processeur p_1 et une copie o_1^2 placée sur le processeur p_2 (figure 8.3a). Dans ce cas, le système peut être vu comme un système constitué de deux composants « o_1^1/p_1 » et « o_1^2/p_2 » en parallèle (figure 8.3b).



a. Distribution et ordonnancement de $\mathcal{A}lg$ b. Système avec redondance en parallèle

FIG. 8.3 – Exemple d'un système avec redondance en parallèle.

Par exemple, la fiabilité $\mathcal{F}_{cal}$ du système en parallèle de la figure 8.3b est calculée par l'équation (8.4) comme suit :

$$\begin{aligned}\mathcal{F}_{cal}^{parallele} &= 1 - \big(1 - e^{-\lambda_1 \mathcal{E}xe(o_1^1,p_1)}\big)\,\big(1 - e^{-\lambda_2 \mathcal{E}xe(o_1^2,p_2)}\big) = 1 - \big(1 - e^{-0.00001\times 1}\big)\,\big(1 - e^{-0.0001\times 2}\big)\\ &= 0.999999998.\end{aligned}$$

Sa durée d'exécution totale est calculée par l'équation (8.1) comme suit :

$$\mathcal{R}tc_{cal} = \max\ \big(Et_{exc}(o_1^1,p_1)\ ,\ Et_{exc}(o_1^2,p_2)\big)\ =\ \max\ \big(1\ ,\ 2\big) = 2$$

Système série/parallèle

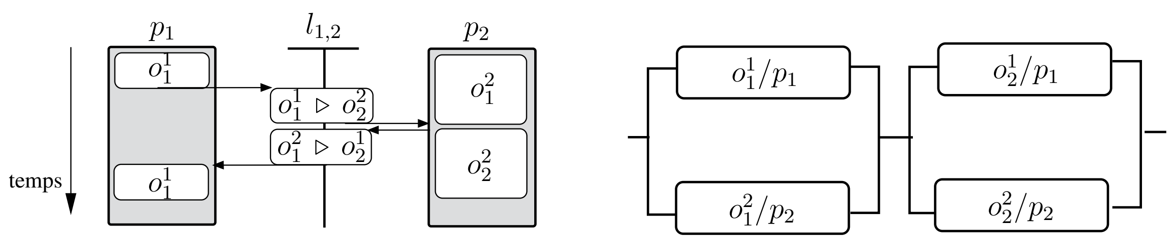
Dans le cas où un système avec redondance logicielle est constitué de composants en série et que chaque composant est constitué de composants en parallèle, il est appelé *système en série/parallèle* [14] :

Définition 45 *Deux éléments sont en série/parallèle si le fonctionnement d'au moins un sous-élément de chacun est suffisant pour assurer le fonctionnement de l'ensemble.*

La probabilité qu'un système en série/parallèle fonctionne correctement est définie par la probabilité qu'au moins un de sous-composants d'un composant en parallèle fonctionnent correctement. Elle est donnée par la formule suivante :

$$\mathcal{F}_{cal}^{serie/parallele} = \prod_{\mathcal{R}ep(o_i) \subset \mathcal{A}lg} \mathcal{F}_{cal}^{parallele}\big(\mathcal{R}ep(o_i)\big) \tag{8.5}$$

La figure 8.4 est un exemple d'un système en série/parallèle, où l'opération o_1 (resp. o_2) de $\mathcal{A}lg$ de la figure 8.2a est répliquée en deux copies : une copie o_1^1 (resp. o_2^1) placée sur le processeur p_1 (resp. p_1) et une copie o_1^2 (resp. o_2^2) placée sur le processeur p_2 (resp. p_2). Si les liens de communications sont supposés sans fautes, alors le système peut être vu comme un système constitué de deux composants en série : C_1 et C_2, et chacun d'entre eux est constitué de deux sous-composants en parallèle : $C1 = \{o_1^1/p_1, o_1^2/p_2\}$ et $C2 = \{o_2^1/p_1, o_2^2/p_2\}$, comme cela est montré sur la figure 8.4b.



a. Distribution et ordonnancement de $\mathcal{A}lg$ b. Système avec redondance en série/parallèle

FIG. 8.4 – Exemple d'un système en série/parallèle sans fautes de liens de communication.

La fiabilité de ce système peut être calculée par l'équation (8.5) comme suit :

$$\begin{aligned}\mathcal{F}_{cal} &= \Big(1 - \big(1 - e^{-\lambda_1 \mathcal{E}xe(o_1^1, p_1)}\big)\big(1 - e^{-\lambda_2 \mathcal{E}xe(o_1^2, p_2)}\big)\Big) \times \\ &\quad \Big(1 - \big(1 - e^{-\lambda_1 \mathcal{E}xe(o_2^1, p_1)}\big)\big(1 - e^{-\lambda_2 \mathcal{E}xe(o_2^2, p_2)}\big)\Big) \\ &= (0.999999998) \times (0.999999998) = 0.9999998\end{aligned}$$

Sa durée d'exécution totale est calculée par l'équation (8.1) comme suit :

$$\mathcal{R}tc_{cal} = \max\ \big(Et_{exc}(o_2^1, p_1)\ ,\ Et_{exc}(o_2^2, p_2)\big)\ =\ \max\ \big(4\ ,\ 4\big) = 4$$

Système de structure quelconque

Dans le cas où le système est soumis aux fautes des liens de communication, le calcul exact de la fiabilité devient complexe, puisque le système peut avoir une structure quelconque (ni série, ni

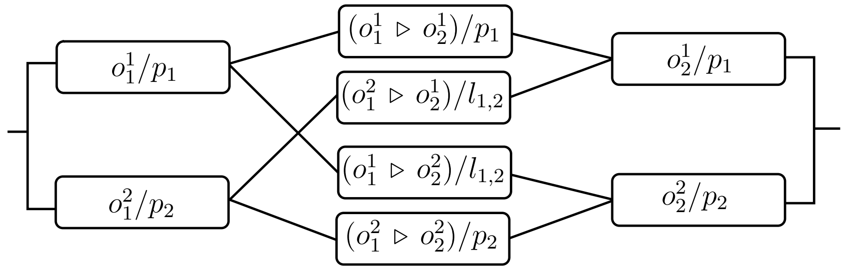


FIG. 8.5 – Exemple d'un système en structure quelconque avec fautes de liens de communication.

parallèle, ni série/parallèle). Par exemple, la figure 8.5 représente la nouvelle structure du système de la figure 8.5. La différence avec la figure 8.5 est qu'ici on a tenu compte des défaillances des liens de communication.

Dans les cas simples, le calcul exact de la fiabilité peut se faire par recherche des combinaisons favorables, c'est-à-dire des combinaisons pour lesquelles le système est dans un état de fonctionnement correct. Par exemple, le système de la figure 8.5 est constitué de huit composants ; quatre opérations, deux communications intra-processeur et deux communications inter-processeurs. Puisque chaque composant est soit dans un état défaillant, soit dans un état non défaillant, il existe 2^8 combinaisons dans ce système, et le système ne fonctionne correctement que dans certaines de ces combinaisons. La fiabilité de ce système est donc la somme des fiabilités de toutes les combinaisons qui mènent le système dans un état de fonctionnement correct [56].

Dans le cas général, si le système comporte n composants, le calcul exact de sa fiabilité nécessite d'évaluer 2^n combinaisons ; ce calcul est donc exponentiel.

Dans ce travail, nous nous intéressons aux méthodes approchées de calcul de la fiabilité des systèmes en structure quelconque. La *transformation de graphe* [56] est l'une des méthodes utilisées dans la littérature pour approximer le calcul de la fiabilité d'un système, que nous utilisons dans notre calcul. Son principe consiste à transformer un système de structure quelconque en un système série/parallèle. L'inconvénient de la transformation de graphe est qu'elle peut augmenter la longueur de la distribution/ordonnancement. La transformation se fait au niveau du graphe d'algorithme $\mathcal{A}lg$, et c'est après le processus de distribution/ordonnancement du nouveau graphe d'algorithme qu'on obtient un système série/parallèle. Elle se fait en deux phases successives (figure 8.6) :

1. **Phase de réplication** : Supposons que N est le nombre de processeurs du graphe d'architecture $\mathcal{A}rc$. Dans cette phase, chaque opération o_i de $\mathcal{A}lg$ est répliquée en au plus N-1 répliques, et chaque réplique envoie ses données de sortie à toutes les répliques de toutes ses successeurs, d'où un nouveau graphe d'algorithme avec redondance. Par exemple, l'opération o_1 (resp. o_2) du graphe d'algorithme $\mathcal{A}lg$ de la figure 8.6a est répliquée en deux copies o_1^1 et o_1^2 (resp. en trois copies o_2^1, o_2^2 et o_2^3) ; c'est l'ensemble $\mathcal{R}ep(o_1)$ (resp. $\mathcal{R}ep(o_2)$), comme cela est montré sur la figure 8.6b.
2. **Phase d'ajout de nouvelles opérations** : Dans cette phase, pour chaque paire d'opérations

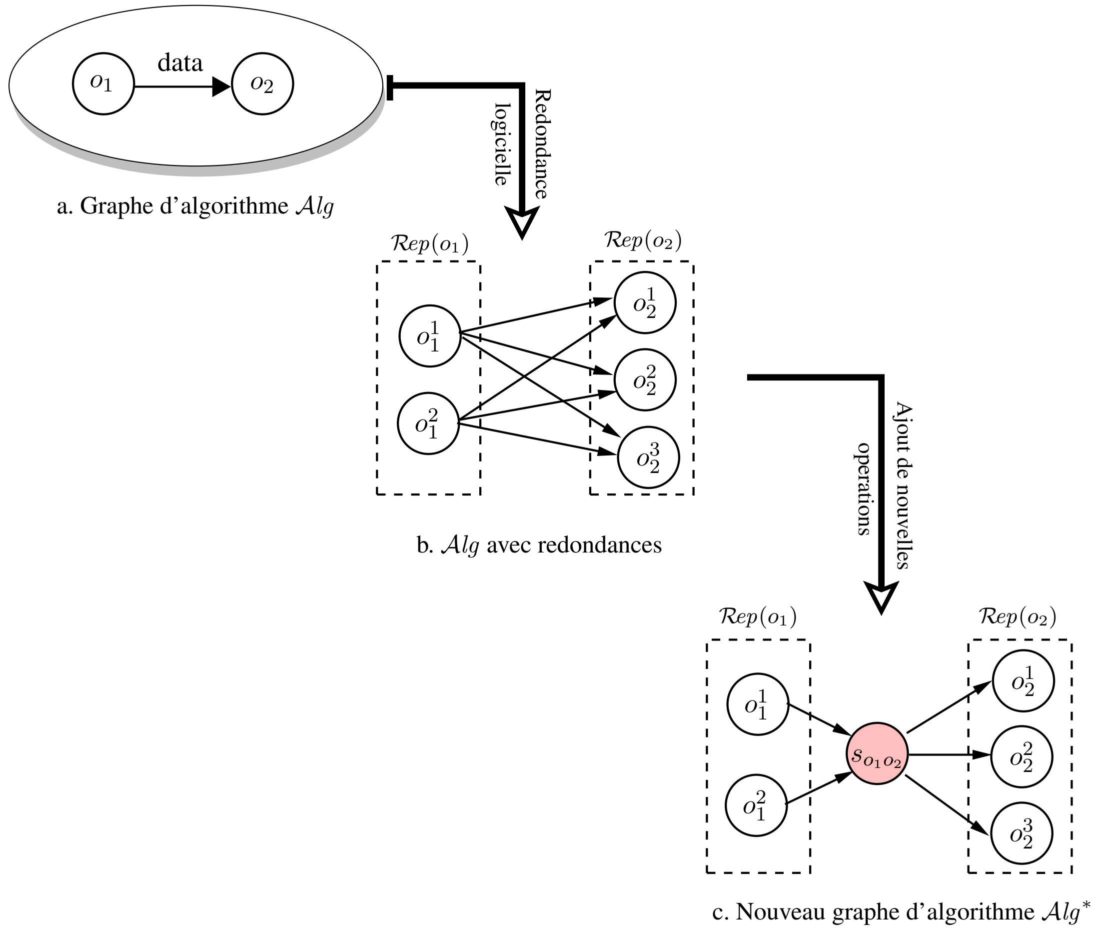


FIG. 8.6 – Transformation de graphe d'algorithme.

dépendantes o_i et o_j de $\mathcal{A}lg$, nous remplaçons, dans le nouveau graphe d'algorithme avec redondance, toutes les dépendances de données entre $\mathcal{R}ep(o_i)$ et $\mathcal{R}ep(o_j)$ par une nouvelle opération de contrôle $s_{o_io_j}$ (cf. section 6.3, page 80), et ensuite nous répliquons la dépendance de données $(o_i \rhd o_j)$ entre les opérations de $\mathcal{R}ep(o_i)$ et $s_{o_io_j}$, et aussi entre $s_{o_io_j}$ et les opérations de $\mathcal{R}ep(o_i)$. Par exemple, les dépendances de données du graphe d'algorithme avec redondance de la figure 8.6b sont toutes remplacées par l'opération de contrôle $s_{o_1o_2}$, un ensemble de dépendances de données entre $\mathcal{R}ep(o_1)$ et $s_{o_1o_2}$, et un autre entre $s_{o_1o_2}$ et $\mathcal{R}ep(o_2)$. Ceci donne le nouveau graphe d'algorithme $\mathcal{A}lg^*$ de la figure 8.6c. Enfin, la distribution/ordonnancement de $\mathcal{A}lg^*$ sur un graphe d'architecure $\mathcal{A}rc$ a la structure série/parallèle de la figure 8.7. Ce nouveau système est composé de quatre composants en série : C_1, C_2, C_3 et C_4.

Remarque 19 *Chaque dépendance de données* $(o_i^l \rhd s_{o_i,o_j})$ *de* $\mathcal{A}lg^*$*, réplique de* $(o_i \rhd o_j)$*, a les mêmes caractéristiques (taille et type de données). De même, chaque dépendance de données*

$(s_{o_i,jj} \triangleright o_j^k)$, réplique de $(o_i \triangleright o_j)$, a les mêmes caractéristiques.

Si on note deux composants C_i et C_j en série par « $C_i \bullet C_j$ », et deux composants en parallèle par « $C_i \| C_j$ ». Le système de la figure 8.7 peut être donc représenté par :

$$\begin{aligned} Sys &= C_1 \bullet C_2 \bullet C_3 \bullet C_4 \\ &= \left[o_1^1 \| o_1^2\right] \bullet \left[(o_1^1 \triangleright s_{o_1o_2}) \| (o_1^2 \triangleright s_{o_1o_2})\right] \bullet s_{o_1o_2} \bullet \\ &\quad \left[\left((s_{o_1o_2} \triangleright o_2^1) \bullet o_2^1\right) \| \left((s_{o_1o_2} \triangleright o_2^2) \bullet o_2^2\right) \| \left((s_{o_1o_2} \triangleright o_2^3) \bullet o_2^3\right)\right] \end{aligned}$$

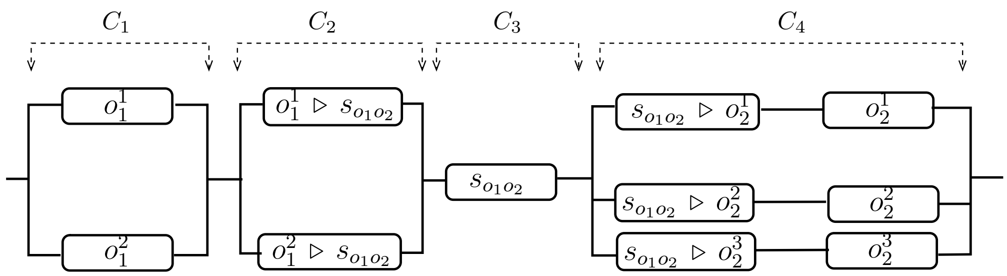


FIG. 8.7 – Nouveau système en série/parallèle après transformation du graphe d'algorithme.

Suivant l'équation (8.5), la probabilité qu'un système (Sys) en structure quelconque, transformé en un système en série/parallèle, fonctionne correctement est calculée par la formule suivante :

$$\mathcal{F}_{cal}^{quelconque} = \prod_{C_i \subset Sys} \mathcal{F}_{cal}^{parallele}(C_i) \times \prod_{s_{o_io_j} \in \mathcal{A}lg^*} \mathcal{F}_{cal}(s_{o_io_j}) \tag{8.6}$$

Remarque 20 (communication intra-processeur fiables) *Nous supposons que la durée d'exécution d'une communication intra-processeur $(o_i \triangleright o_j)$ est nulle, d'où :*

$$\mathcal{F}_{cal}(o_i \triangleright o_j) = exp^{-\mu_{ms}\mathcal{E}xe((o_i \triangleright o_j), l_{ms})} = exp^{-\mu_{ms} \times 0} = 1$$

Remarque 21 (opérations de contrôle fiables) *Nous supposons que la durée d'exécution d'une opération de contrôle $s_{o_io_j}$ est nulle, d'où :*

$$\mathcal{F}_{cal}(s_{o_io_j}) = exp^{-\lambda_p \mathcal{E}xe(s_{o_io_j}, p)} = exp^{-\lambda_p \times 0} = 1$$

Suivant cette dernière remarque, on a :

$$\mathcal{F}_{cal}^{quelconque} = \prod_{C_i \subset Sys} \mathcal{F}_{cal}^{parallele}(C_i) \tag{8.7}$$

La fiabilité du nouveau système de la figure 8.7 peut être calculée par cette équation comme suit :

$$\mathcal{F}_{cal}^{quelconque} = \mathcal{F}_{cal}^{parallele}(C_1) \times \mathcal{F}_{cal}^{parallele}(C_2) \times \mathcal{F}_{cal}^{parallele}(C_3) \times \mathcal{F}_{cal}^{parallele}(C_4)$$

où,

$$\begin{aligned}
\mathcal{F}_{cal}^{parallele}(C_1) &= 1-(1-e^{-\lambda_1 \mathcal{E}xe(o_1^1,p_1)})\,(1-e^{-\lambda_2 \mathcal{E}xe(o_1^2,p_2)}) \\
\mathcal{F}_{cal}^{parallele}(C_2) &= 1-(1-e^{-\mu_{xy} \mathcal{E}xe(o_1^1 \triangleright s_{o_1 o_2},l_{xy})})(1-e^{-\mu_{zw} \mathcal{E}xe(o_1^2 \triangleright s_{o_1 o_2},l_{zw})}) \\
\mathcal{F}_{cal}^{parallele}(C_3) &= \mathcal{F}_{cal}(s_{o_i o_j}) = 1 \\
\mathcal{F}_{cal}^{parallele}(C_4) &= 1-(1-e^{-\mu_{xy} \mathcal{E}xe(s_{o_1 o_2} \triangleright o_2^1,l_{xy})-\lambda_i \mathcal{E}xe(o_2^1,p_i)}) \times \\
&\quad\ (1-e^{-\mu_{zw} \mathcal{E}xe(s_{o_1 o_2} \triangleright o_2^2,l_{zw})-\lambda_j \mathcal{E}xe(o_2^2,p_j)}) \times \\
&\quad\ (1-e^{-\mu_{st} \mathcal{E}xe(s_{o_1 o_2} \triangleright o_2^3,l_{st})-\lambda_k \mathcal{E}xe(o_2^3,p_k)})
\end{aligned}$$

8.2.3 Deux critères antagonistes

La redondance logicielle présente un inconvénient de surcoût en longueur de la distribution/ordonnancement $\mathcal{R}tc_{cal}$, mais elle apporte un avantage d'augmentation de la fiabilité de cette distribution/ordonnancement $\mathcal{F}_{cal}$. D'où, un *problème antagoniste*, puisque plus de redondance implique un surcoût en longueur $\mathcal{R}tc_{cal}$ mais une meilleure fiabilité $\mathcal{F}_{cal}$, tandis que moins de redondance implique une meilleure longueur $\mathcal{R}tc_{cal}$ mais une fiabilité réduite $\mathcal{F}_{cal}$. Donc, la méthodologie AAA-F que nous proposons doit trouver un compromis entre $\mathcal{R}tc_{cal}$ et $\mathcal{F}_{cal}$ pour résoudre ce problème antagoniste, c'est-à-dire qu'elle doit déterminer le nombre de répliques pour chaque opération o_i d'un graphe d'algorithme $\mathcal{A}lg$ en fonction des deux critères $\mathcal{R}tc_{obj}$ et $\mathcal{F}_{obj}$. La solution que nous proposons est détaillée dans la section suivante.

8.3 Heuristique de distribution et d'ordonnancement bi-critères

Nous présentons, dans cette section, une heuristique de distribution/ordonnancement qui permet de résoudre le problème 5. Notre heuristique est un algorithme de construction progressive de type glouton [11]. Puisque nous utilisons la redondance logicielle pour augmenter la fiabilité d'une distribution/ordonnancement et que le nombre des répliques de chaque opération o_i de $\mathcal{A}lg$ est différent d'un composant à un autre, l'heuristique doit à chaque étape (n) :

- *calculer* le nombre de répliques pour chaque opération o_i de $\mathcal{A}lg^{(n)}$,
- puis, *transformer* le graphe d'algorithme $\mathcal{A}lg^{(n-1)}$ en un nouveau graphe $\mathcal{A}lg^{(n)}$,
- et enfin, *distribuer/ordonnancer* les répliques de o_i et aussi leurs opérations de contrôle.

8.3.1 Principe de l'heuristique

Le but principal de notre heuristique de distribution/ordonnancement est de satisfaire les deux critères de fiabilité $\mathcal{F}_{obj}$ et de temps réel $\mathcal{R}tc_{obj}$, c'est-à-dire maximiser la fiabilité $\mathcal{F}_{cal}$ et minimiser la longueur $\mathcal{R}tc_{cal}$ de la distribution/ordonnancement générée. L'utilisation de la redondance logicielle présente un problème antagoniste (section 8.2.3), donc l'heuristique que nous proposons doit déterminer à chaque étape (n) le nombre de répliques pour chaque opération o_i qui doit maximiser ${\mathcal{F}_{cal}}^{(n)}$ et minimiser ${\mathcal{R}tc_{cal}}^{(n)}$. ${\mathcal{F}_{cal}}^{(n)}$ (resp. ${\mathcal{R}tc_{cal}}^{(n)}$) désigne la fiabilité (resp. la longueur) de la distribution/ordonnancement après avoir répliqué o_i et placé ses répliques à l'étape (n) de l'heuristique.

Notons, que la fiabilité $\mathcal{F}_{cal}$ calculée par l'équation 8.7 est une borne inférieure de la fiabilité exacte $\mathcal{F}_{exact}$ [56], d'où $\mathcal{F}_{exact} \geq \mathcal{F}_{cal}$, et que dans l'heuristique on vérifie à chaque étape (n) que $\mathcal{F}_{cal} \geq \mathcal{F}_{obj}$, ce qui donne $\mathcal{F}_{exact} \geq \mathcal{F}_{cal} \geq \mathcal{F}_{obj}$. Donc, si l'heuristique trouve une distribution/ordonnancement de fiabilité $\mathcal{F}_{cal}$ supérieure à $\mathcal{F}_{obj}$, alors la fiabilité exacte $\mathcal{F}_{exact}$ de cette distribution/ordonnancement satisfait aussi $\mathcal{F}_{obj}$.

8.3.1.1 Critère sur la longueur de la distribution/ordonnancement

Afin de minimiser la longueur de la distribution/ordonnancement, l'heuristique est basée sur une fonction de coût, appelée *gain en longueur*, qui permet d'introduire un ordre d'exécution entre les opérations. Elle calcule pour chaque opération o_i le gain en longueur de la distribution/ordonnancement résultant du placement de ses k' répliques sur un ensemble P de k' processeurs. Elle est définie suivant la figure 8.8a par la formule suivante :

$$\mathcal{G}^{(n)} = \frac{{\mathcal{R}tc_{cal}}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}) - {\mathcal{R}tc_{cal}}^{(n-1)}}{\mathcal{R}tc_{obj} - {\mathcal{R}tc_{cal}}^{(n-1)}} \tag{8.8}$$

On normalise [27] le gain en longueur ${\mathcal{R}tc_{cal}}^{(n)} - {\mathcal{R}tc_{cal}}^{(n-1)}$ par rapport à la marge restante pour ne pas dépasser l'objectif $\mathcal{R}tc_{obj} - {\mathcal{R}tc_{cal}}^{(n-1)}$ afin que les deux critères aient le même ordre de grandeur [5]. On procédera de même pour le critère de fiabilité.

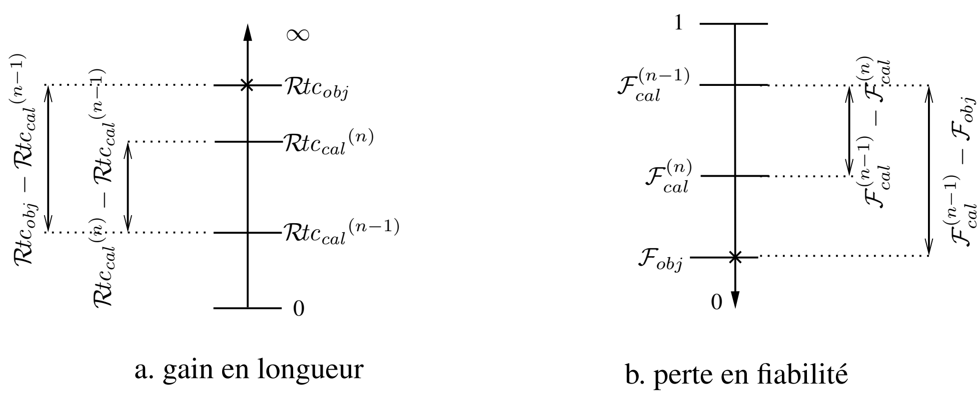


FIG. 8.8 – Calcul du gain en longueur et de la perte en fiabilité.

8.3.1.2 Critère sur la fiabilité de la distribution/ordonnancement

Afin de maximiser la fiabilité de la distribution/ordonnancement, l'heuristique est basée sur une fonction de coût, appelée *perte en fiabilité*, qui permet d'introduire un ordre d'exécution entre les opérations. Elle calcule pour chaque opération o_i la perte en fiabilité de la distribution/ordonnancement résultant du placement de ses k' répliques sur un ensemble P de k' processeurs. Elle est définie suivant la figure 8.8b par la formule suivante :

$$\mathcal{P}^{(n)} = \frac{\mathcal{F}_{cal}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}) - \mathcal{F}_{cal}^{(n-1)}}{\mathcal{F}_{obj} - \mathcal{F}_{cal}^{(n-1)}} \tag{8.9}$$

Ici, aussi la perte en fiabilité est normalisée par rapport à la marge restante.

8.3.1.3 Compromis entre les deux critères

La solution que nous proposons pour résoudre notre problème bi-critères antagoniste consiste à chercher, à chaque étape (n) de l'heuristique, un *compromis* [27] entre la fonction de perte en fiabilité $\mathcal{P}^{(n)}$ et la fonction de gain en longueur $\mathcal{G}^{(n)}$, qui doit à la fois sélectionner la meilleure opération o_i de $\mathcal{A}lg$ à placer/ordonnancer, et calculer le nombre de ses répliques, tout en respectant $\mathcal{R}tc_{obj}$ et $\mathcal{F}_{obj}$.

La recherche d'un compromis consiste à choisir pour chaque opération o_i le meilleur couple $(\mathcal{G}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}), \mathcal{P}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}))$ qui satisfait les deux critères $\mathcal{R}tc_{obj}$ et $\mathcal{F}_{obj}$ à la fois. Chaque couple $(\mathcal{G}^{(n)}, \mathcal{P}^{(n)})$ peut être interprété comme un point du plan avec $\mathcal{G}^{(n)}$ en abscisse et $\mathcal{P}^{(n)}$ en ordonnée.

Principe 10 *Étant donné que $\mathcal{P}^{(n)}$ et $\mathcal{G}^{(n)}$ sont normalisées, donc :*

1. *Si ($\mathcal{P}^{(n)}>1$), alors l'objectif ($\mathcal{F}_{cal}^{(n)}<\mathcal{F}_{obj}$) n'est pas satisfait.*
2. *De même, si ($\mathcal{G}^{(n)}>1$), alors l'objectif ($\mathcal{R}tc_{cal}{}^{(n)}<\mathcal{R}tc_{obj}$) n'est pas satisfait.*

On utilise ces deux propriétés de $\mathcal{G}^{(n)}$ et $\mathcal{P}^{(n)}$ dans l'heuristique pour détecter qu'un des critères ne peut être satisfait.

Suivant ce principe 10, les couples acceptables pour chaque opération o_i se trouvent dans le carré $[0,1] \times [0,1]$, comme cela est montré sur la figure 8.9. Par exemple, les couples $(o_1, \{p_2\})$ et $(o_1, \{p_1, p_3\})$ satisfont les deux critères, tandis que le couple $(o_1, \{p_1, p_2, p_3\})$ ne satisfait pas le critère $\mathcal{R}tc_{obj}$.

Afin de choisir le meilleur couple $(\mathcal{G}^{(n)}, \mathcal{P}^{(n)})^{best}$ qui satisfait les deux critères à la fois, nous proposons de projeter orthogonalement les points $(\mathcal{G}^{(n)}, \mathcal{P}^{(n)})$ sur la droite $\mathcal{G}^{(n)} = \tan(\theta)\ \mathcal{P}^{(n)}$, comme cela est illustré sur la figure 8.9 pour les deux couples $(o_1, \{p_1, p_3\})$ et $(o_1, \{p_2\})$. Le meilleur couple est donc celui qui correspond à la *projection orthogonale minimale*. Dans l'ensemble de la figure 8.9, il est préférable de ne pas répliquer o_1 sur les deux processeurs $\{p_1, p_3\}$, et donc le meilleur placement $(o_1, \{p_1, p_3\})^{best}$ qui satisfait les deux critères à la fois est de placer o_1 sur p_1. Ainsi, la valeur de chaque projection orthogonale d'un couple est calculée, suivant la figure 8.9, par une fonction de coût, appelée *fonction de compromis bi-critères*. Elle est donnée par la formule suivante :

$$\mathcal{C}omp^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}) = \cos(\theta)\ \mathcal{P}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\})\ +\ \sin(\theta)\ \mathcal{G}^{(n)}(o_i, \{p_1, \ldots, p_{k'}\}) \tag{8.10}$$

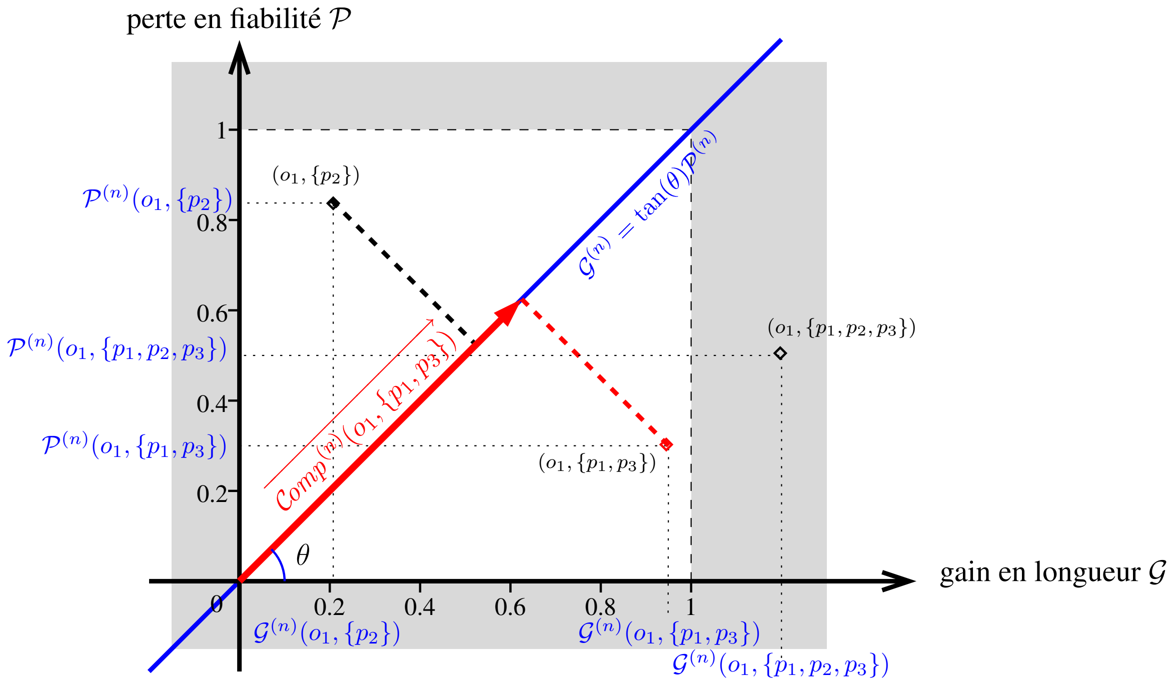


FIG. 8.9 – Calcul d'un compromis entre $\mathcal{G}^{(n)}$ et $\mathcal{P}^{(n)}$.

Le paramètre θ, qui est à l'angle de la droite avec l'horizontale, est un arbitre entre $\mathcal{P}^{(n)}$ et $\mathcal{G}^{(n)}$. Il permet de privilégier, pour ($\theta \neq 45^\circ$), une des deux fonctions par rapport à l'autre. Si à l'étape (n) l'heuristique échoue à satisfaire un des deux critères, elle peut être re-exécutée à partir d'une étape précédente $(n-m)$ en changeant la valeur de θ :

$$\theta = \begin{cases} \theta + \Delta\theta & \text{Si} \quad \mathcal{F}_{cal}^{(n)} < \mathcal{F}_{obj} \\ \theta - \Delta\theta & \text{Si} \quad \mathcal{R}tc_{cal}{}^{(n)} > \mathcal{R}tc_{obj} \end{cases} \tag{8.11}$$

où, $\Delta\theta$ est un paramètre strictement positif de l'heuristique, et $1 \leq m \leq n$.

8.3.2 Présentation de l'algorithme de l'heuristique

L'algorithme de notre heuristique de distribution/ordonnancement se divise en six phases : une phase d'initialisation, une phase de transformation de graphe d'algorithme, une phase de sélection, une phase de distribution/ordonnancement, puis une phase de vérification des critères, et enfin une phase de mise à jour.

ALGORITHME

- ***Entrées =*** *Graphe d'algorithme $\mathcal{A}lg$, graphe d'architecture $\mathcal{A}rc$, caractéristiques d'exécution $\mathcal{E}xe$, contraintes matérielles $\mathcal{D}is$, critère de fiabilité $\mathcal{F}_{obj}$, critère de temps réel $\mathcal{R}tc_{obj}$, un paramètre $\Delta\theta$ pour le retour-arrière en cas d'échec de l'heuristique, et une borne k pour limiter le nombre maximum des répliques d'une même opération ;*
- ***Sortie =*** *Une distribution et un ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$ de fiabilité $\mathcal{F}_{cal} \geq \mathcal{F}_{obj}$ et de longueur $\mathcal{R}tc_{cal} \leq \mathcal{R}tc_{obj}$, ou un message d'échec ;*

INITIALISATION

Initialiser la liste des opérations candidates, et la liste des opérations déjà placées :

$O^{(1)}_{cand}$:={*opérations de* $\mathcal{A}lg$ *sans prédécesseurs*} ;
$O^{(1)}_{fin} := \emptyset$;

Initialiser un ensemble $\mathcal{C}^k$ de toutes les combinaisons arbitraires d'au plus k' processeurs ($k' \leq k$) ;

BOUCLE DE DISTRIBUTION ET D'ORDONNANCEMENT

Tant que $O^{(n)}_{cand} \neq \emptyset$ ***faire***

TRANSFORMATION DU GRAPHE D'ALGORITHME

- ***Pour chaque*** *opération o_{cand} de $O^{(n)}_{cand}$ qui n'est pas encore répliquée* ***faire***
  - *Répliquer o_{cand} en k répliques ; soit $\mathcal{R}ep(o_{cand})$ l'ensemble de ses répliques ;*
  - *Remplacer chaque dépendance de données ($o_{pred} \rhd o_{cand}$) par une opération de contrôle $s_{o_{pred}o_{cand}}$; ensuite, répliquer la dépendance ($o_{pred} \rhd o_{cand}$) en k répliques entre $\mathcal{R}ep(o_{pred})$ et $s_{o_{pred}o_{cand}}$, et en k répliques entre $s_{o_{pred}o_{cand}}$ et $\mathcal{R}ep(o_{cand})$;*
- ***Fin pour chaque***

SÉLECTION

- ***Pour chaque*** *combinaison $P_{k'}$ de k' processeurs de $\mathcal{C}^k$* ***faire***
  - *Soit $\mathcal{O}^{k'}_{cand}$ un ensemble de k' répliques de o_{cand} ;*
  - *Calculer pour chaque ensemble $\mathcal{O}^{k'}_{cand}$ la fonction :*

$$\mathcal{C}omp^{(n)}(\mathcal{O}^{k'}_{cand}, P_{k'}) := \cos(\theta)\mathcal{P}^{(n)}(\mathcal{O}^{k'}_{cand}, P_{k'}) + \sin(\theta)\mathcal{G}^{(n)}(\mathcal{O}^{k'}_{cand}, P_{k'}) \tag{8.12}$$

- ***Fin pour chaque***
- *Sélectionner pour chaque opération candidate o_{cand} de $O^{(n)}_{cand}$ le couple ($\mathcal{O}^{k'}_{cand}$,$P^{best}_{k'}$) qui minimise l'équation (8.12) :*

$$\mathcal{C}omp^{(n)}(\mathcal{O}^{k'}_{cand}, P^{best}_{k'}) := \min\left(\mathcal{C}omp^{(n)}(\mathcal{O}^{k'}_{cand}, P_i)\right)$$

- *Sélectionner, parmi les couples ($\mathcal{O}^{k'}_{cand}$,$P^{best}_{k'}$), le meilleur couple ($\mathcal{O}^{k'}_{best}$,$P^{best}_{k'}$) qui maximise l'équation 8.12 :*

$$\mathcal{C}omp^{(n)}(\mathcal{O}^{k'}_{best}, P^{best}_{k'}) := \max\left(\mathcal{C}omp^{(n)}(\mathcal{O}^{k'}_{cand}, P_i)\right)$$

DISTRIBUTION ET ORDONNANCEMENT

- *Supprimer du graphe d'algorithme $\mathcal{A}lg^{(n)}$ $(k-k')$ opérations répliques de o_{best} ; leurs dépendances de données sont également supprimées ;*
- *Placer et ordonnancer toutes les opérations de contrôle $s_{o_{pred}o_{best}}$ de o_{best} ; les communications induites par ces placements sont également placées/ordonnancées ;*
- *Placer et ordonnancer toutes les k' opérations restantes de o_{cand} sur les k' processeurs $P_{k'}^{best}$; les communications induites par ce placement sont également placées/ordonnancées ;*

VÉRIFICATION DES CRITÈRES

- **si** $\left(\mathcal{F}_{cal}^{(n)}(\mathcal{O}_{best}^{k'}, P_{k'}^{best}) < \mathcal{F}_{obj}^{(n)}\right)$ **alors**
  - $\theta := \theta \, + \, \Delta\theta$;
  - **si** *($\theta > 90\,^{\circ}$)* **alors** *« message d'échec » ;*
  - **sinon** *Re-exécuter l'heuristique à partir de l'étape (m) qui maximise l'équation : $\mathcal{P}^{(m)}/\mathcal{G}^{(m)}$;*
- **si** $\left(\mathcal{R}tc_{cal}{}^{(n)}(\mathcal{O}_{best}^{k'}, P_{k'}^{best}) > \mathcal{R}tc_{obj}{}^{(n)}\right)$ **alors**
  - $\theta := \theta \, - \, \Delta\theta$;
  - **si** *($\theta < 0\,^{\circ}$)* **alors** *« message d'échec » ;*
  - **sinon** *Re-exécuter l'heuristique à partir de l'étape (m) qui maximise l'équation : $\mathcal{G}^{(m)}/\mathcal{P}^{(m)}$;*

MISE À JOUR

- *Mettre à jour la liste des opérations candidates et déjà placées :*
  $O_{fin}^{(n+1)} := O_{fin}^{(n)} \cup O_{best}^{k'}$;
  $O_{cand}^{(n+1)} := O_{cand}^{(n)} - O_{best}^{k'} \cup \left\{ o \in succ(o_{best}) \mid pred(o) \subseteq O_{fin}^{(n+1)} \right\}$;

Fin tant que

FIN DE L'ALGORITHME

8.3.2.1 Phase d'initialisation

Cette phase consiste à initialiser la liste des opérations candidates $O_{cand}^{(1)}$ avec les opérations sans prédécesseur. Les seules opérations implantables à cette première étape de l'heuristique sont les opérations d'entrées (capteurs). De plus, la liste des opérations déjà placées $O_{fin}^{(1)}$ est vide. Puisque chaque opération sera répliquée en k' répliques ($k' \leq k$), et que ces répliques devront choisir k' meilleurs processeurs, il est intéressant d'avoir une liste $\mathcal{C}^k$ constituée de toutes les combinaisons arbitraires d'au plus k processeurs afin d'accélérer ce choix. La borne k est le nombre maximal des répliques autorisées pour chaque opération, ceci afin de réduire la complexité de l'heuristique. Cette borne est donnée par l'utilisateur.

8.3.2.2 Phase de transformation du graphe d'algorithme

Nous appliquons les règles de transformations de la figure 8.6 en répliquant chaque opération candidates (et uniquement celles-la) k fois. Par la suite, en ordonnant les opérations candidates selon la fonction de compromis $\mathcal{C}omp^{(n)}$, une opération sera choisie et répliquée en k' exemplaires, avec $k'<k$. À ce moment, on effacera les $k-k'$ répliques non utilisées, ceci afin de diminuer la complexité de l'heuristique.

8.3.2.3 Phase de sélection

Dans cette phase, la candidate la plus urgente o^k_{best} est sélectionnée parmi toutes les opérations candidates pour être placée et ordonnancée. La règle de sélection choisie repose sur la fonction de compromis bi-critères de l'équation (8.12), qui permet de sélectionner la meilleure candidate o_{best} qui maximise cette fonction, ainsi que les k' meilleurs processeurs $P^{best}_{k'}$ qui minimisent cette fonction.

8.3.2.4 Phase de distribution/ordonnancement

Puisque le nombre k' de répliques de la meilleure candidate o_{best}, calculé par la phase précédente, peut être inférieur à k, $(k-k')$ répliques de o_{best} seront supprimées de la liste des candidates $O^{(n)}_{cand}$ et aussi du nouveau graphe d'algorithme $\mathcal{A}lg^{(n)}$. Ensuite, avant de placer et ordonnancer les k' répliques restantes de o_{best} sur les processeurs de $P^{best}_{k'}$, ses opérations de contrôle $s_{o_{pred}o_{best}}$ seront tout d'abord placées et ordonnancées.

8.3.2.5 Phase de vérification des critères

Après avoir placé et ordonnancé les répliques de o_{best} et leurs opérations de contrôle, cette phase consiste à vérifier si les deux critères de fiabilité et de temps réel sont respectés ou non. En cas de non respect d'un critère à une étape (n), l'heuristique peut être re-exécutée à partir d'une étape (m) avec une nouvelle valeur pour θ, déterminée par l'équation (8.11). L'étape (m) est sélectionnée parmi les étapes de (1) à $(n-1)$ telle que l'écart entre la fonction de la fiabilité $\mathcal{G}^{(m)}$ et la fonction de temps réel $\mathcal{P}^{(m)}$ soit maximal.

8.3.2.6 Phase de mise à jour

Cette phase consiste à mettre à jour tout d'abord la liste des opérations déjà placées et ensuite la liste des opérations candidates. De plus, toutes les k' répliques de o_{best} sont supprimées de la liste des candidates $O^{(n)}_{cand}$, et enfin les nouvelles opérations ajoutées à cette liste sont les opérations de $\mathcal{A}lg^{(n)}$ qui ont toutes leurs prédécesseurs dans la liste des opérations déjà placées $O^{(n+1)}_{fin}$.

8.4 Simulations

Afin d'évaluer l'heuristique de distribution/ordonnancement de AAA-F, avons comparé ses performances avec l'heuristique HTR de distribution/ordonnancement de SYNDEX présentée dans la section 2.6 (page 27). HTR est basée sur une fonction de coût à un seul critère à minimiser, qui est le critère temps réel $\mathcal{R}tc_{obj}$, et elle ne prend pas en compte le critère de la fiabilité $\mathcal{F}_{obj}$. L'objectif général de cette simulation est de comparer la longueur et la fiabilité de la distribution/ordonnancement générées par AAA-F et HTR. Enfin, nous avons implementé l'heuristique AAA-F dans l'outil SYNDEX.

8.4.1 Les paramètres de simulation

Nous avons appliqué les deux heuristiques AAA-F et HTR à un ensemble de graphes d'algorithmes générés aléatoirement et à un seul graphe d'architecture de P=5 processeurs complètement connecté par des liens point-à-point. Afin de générer ces graphes aléatoires, nous avons fait varier le nombre N d'opérations dans notre générateur aléatoire de graphes (présenté dans le chapitre 10) : $N = 20, 40, 60, 80, 100$.

Les coûts d'exécution des opérations sont tirés aléatoirement entre 15 et 25, et les coûts des communications sont tirés aléatoirement entre $15 \times CCR$ et $25 \times CCR$. De plus, les taux de défaillance des processeurs (resp. liens de communication) sont tirés aléatoirement entre 5×10^{-5} et 10^{-4} (resp. entre 15×10^{-5} et 3×10^{-4}).

8.4.2 Les résultats

Nous avons tracé dans les deux figures 8.10 et 8.11 respectivement la moyenne en longueur et en fiabilité de la distribution/ordonnancement de 50 graphes aléatoires pour P=5, CCR=1 et $N = 20, \ldots, 100$.

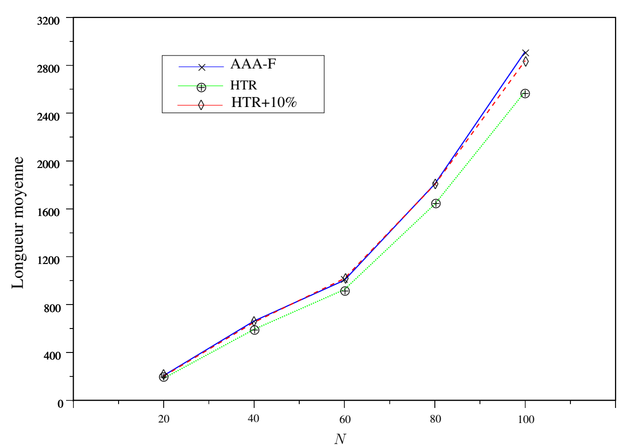


FIG. 8.10 – Comparaison en longueur de la distribution/ordonnancement.

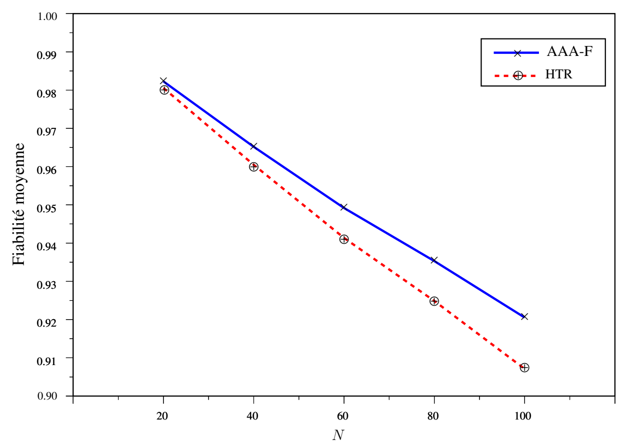


FIG. 8.11 – Comparaison en fiabilité de la distribution/ordonnancement.

Dans chaque distribution/ordonnancement d'un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$, le critère de fiabilité $\mathcal{F}_{obj}$ donné comme objectif à maximiser pour AAA-F est la fiabilité $\mathcal{F}_{cal}$ calculée par HTR. Par contre, le critère temps réel $\mathcal{R}tc_{obj}$ donné comme objectif à minimiser pour AAA-F est la longueur de la distribution/ordonnancement $\mathcal{R}tc_{cal}$ calculée par HTR augmentée de 10% (également tracée sur la figure 8.10). Ceci dans le but d'éviter le retour-arrière en cas d'échec de l'heuristique de satisfaire le cirtère temps réel $\mathcal{R}tc_{obj}$.

La figure 8.10 montre que, dans la plupart des cas, AAA-F satisfait la contrainte temps réel de HTR+10%. Par contre, la figure 8.11 montre que dans tous les cas AAA-F trouve des distributions/ordonnancements plus fiables que HTR.

8.5 Conclusion

Nous avons présenté dans ce chapitre une méthodologie AAA-F originale, basée sur la théorie d'ordonnancement, pour la conception des systèmes réactifs fiables. Plus précisément, AAA-F peut générer une distribution/ordonnancement d'un algorithme sur une architecture qui respecte deux critères antagonistes, qui sont la maximisation de la fiabilité et la minimisation de la longueur de cette distribution/ordonnancement. Étant donné que le calcul de la fiabilité, en cas général, est exponentiel, AAA-F utilise un calcul approché basé sur la transformation de graphe d'algorithme. Enfin, l'algorithme de AAA-F étant heuristique, il se peut qu'il ne trouve aucune solution valide alors qu'il en existe une.

Troisième partie

Développement logiciel

Chapitre 9

Le logiciel SYNDEX

Résumé

Ce chapitre présente le logiciel SYNDEX implantant les trois méthodologies AAA-TP, AAA-TB et AAA-F.

9.1 Présentation générale

Les trois méthodologies AAA-TP, AAA-TB et AAA-F ont été implantées dans un logiciel interactif de développement pour applications temps réel, appelé SYNDEX[1]. Il permet de générer, à partir d'une spécification d'un graphe d'algorithme $\mathcal{A}lg$, d'un graphe d'architecture $\mathcal{A}rc$, d'un ensemble de caractéristiques d'exécution $\mathcal{E}xe$ de $\mathcal{A}lg$ sur $\mathcal{A}rc$, de contraintes matérielles $\mathcal{D}is$ et d'une contrainte temps réel $\mathcal{R}tc$, une distribution et un ordonnancement de $\mathcal{A}lg$ sur $\mathcal{A}rc$. Ensuite, si toutes les contraintes sont satisfaites, alors il génère un exécutif sur mesure pour cette application. La figure 9.1 présente la capture d'écran de la version 6.7.0 de SYNDEX.

9.2 Spécification

9.2.1 Graphe d'algorithme

L'algorithme $\mathcal{A}lg$ est modélisé par un graphe flot de donnée, où les nœuds sont les opérations de calcul de l'algorithme et les arcs sont les dépendances de données entre opérations. La figure 9.2 présente la capture d'écran de SYNDEX, correspondant au graphe d'algorithme. On distingue sur cette figure six opérations ; chaque opération est représentée par un rectangle. Les trois rectangles bleu correspondent à trois opérations de calcul (A, B et C), le rectangle rouge correspond à une opération de sortie O, et les deux autres rectangles correspondent à deux opérations d'entrée (I1 et I2). Chaque opération o_i possède un ensemble de ports d'entrée qui correspondent à la liste

[1] SYNDEX = **Syn**chronized **D**istributed **Ex**ecutive. http ://www-rocq.inria.fr/syndex.

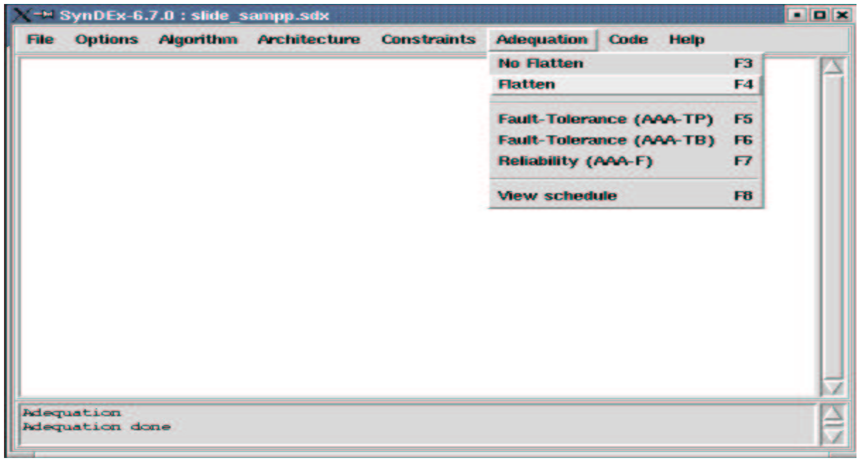


FIG. 9.1 – Le logiciel SYNDEX.

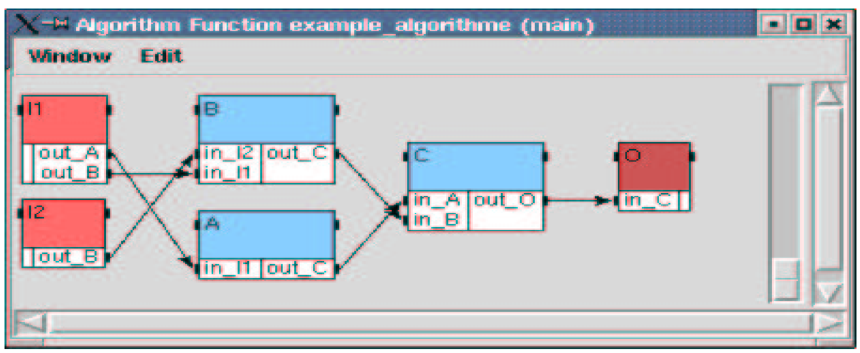


FIG. 9.2 – Exemple d'un graphe d'algorithme $\mathcal{A}lg$.

d'appel de la fonction de o_i, et un ensemble de ports de sortie qui correspondent à la liste des résultats calculés par cette fonction. Par exemple, l'opération C peut être représentée par le code informatique suivant :

```
function C <in_A, in_B>
begin
 ...
 ...
 return <out_O>;
end;
```

On distingue aussi sur cette figure six arcs, et donc six dépendances de données. Chaque opération est connectée à un arc via un port ; par exemple, l'opération A est connectée à l'arc (A $\triangleright$ C) via le port `out_C`.

9.2.2 Graphe d'architecture

L'architecture $\mathcal{A}rc$ est modélisée par un graphe non orienté, où les nœuds désignent les processeurs et les mémoires SAM, et les arêtes désignent les liaisons physiques. La figure 9.3 présente la capture d'écran de SYNDEX, correspondant au graphe d'architecture.

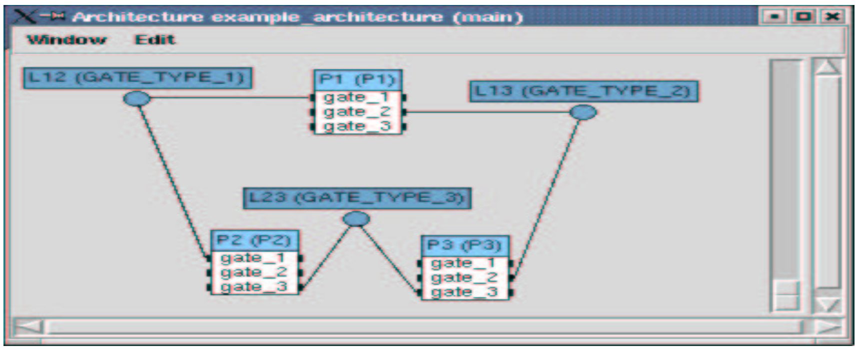


FIG. 9.3 – Exemple d'un graphe d'architecture $\mathcal{A}rc$.

On distingue sur cette figure trois processeurs et trois mémoires SAM de type point-à-point ; tous représentés par des rectangles. Les trois rectangles en blanc et bleu correspondent aux trois processeurs P1, P2 et P3, et les trois rectangles en bleu correspondent aux trois mémoires SAM point-à-point L12, L13 et L23. Chaque processeur P_i possède un ensemble de ports qui correspondent à ses communicateurs. Par exemple, le processeur P1 possède trois communicateurs gate_1, gate_2 et gate_3 ; deux de ses communicateurs (gate_1, gate_2) sont connectés à des mémoires SAM.

9.2.3 Caractéristiques d'exécution

SYNDEX présente un environnement interactif qui permet de spécifier facilement les caractéristiques d'exécution $\mathcal{E}xe$ de chaque composant logiciel du graphe d'algorithme $\mathcal{A}lg$ sur chaque composant matériel du graphe d'architecture. Par exemple, sur la figure 9.4, qui présente la capture d'écran de SYNDEX, sont affichés les coûts d'exécution de toutes les opérations de $\mathcal{A}lg$ sur le processeur P1 de $\mathcal{A}rc$.

9.3 Heuristiques de distribution/ordonnancement temps réel

En plus de l'heuristique implantée dans SYNDEX, nous avons étendu SYNDEX à la tolérance aux fautes et à la fiabilité. Nous avons donc implanté dans SYNDEX les heuristiques des trois méthodologies AAA-TP, AAA-TB et AAA-F. Nous avons de plus implanté l'heuristique de Hashimoto [39] pour faire des simultations comparatives. L'ensemble de ces développements logiciels représentent 2800 lignes de code OCaml.

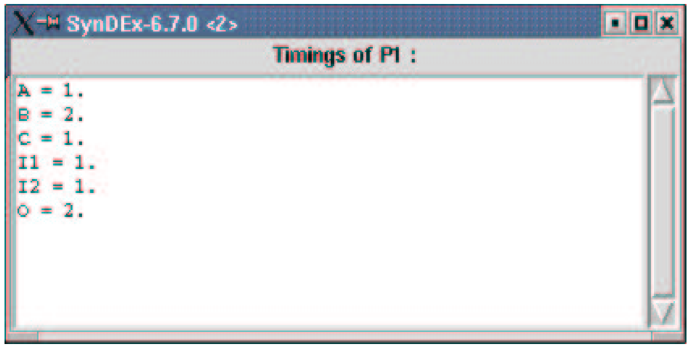


FIG. 9.4 – Spécification des coûts d'exécution $\mathcal{E}xe$.

9.3.1 Heuristique uni-critère de la méthodologie AAA

L'heuristique de la méthodologie AAA implantée dans SYNDEX est celle présentée dans la section 2.6 (page 27). Elle a un seul objectif qui est la minimisation de la longueur de la distribution/ordonnancement afin de satisfaire la contrainte temps réel $\mathcal{R}tc$. L'application de cette heuristique sur le graphe d'algorithme $\mathcal{A}lg$ de la figure 9.2 et le graphe d'architecture $\mathcal{A}rc$ de la figure 9.3 génère la distribution/ordonnancement de la figure 9.5. La longueur $\mathcal{R}tc_{cal}$ de cette distribution/ordonnancement est égale à 7, et la fiabilité $\mathcal{F}_{cal}$ de cette distribution/ordonnancement est égale à 0.999947.

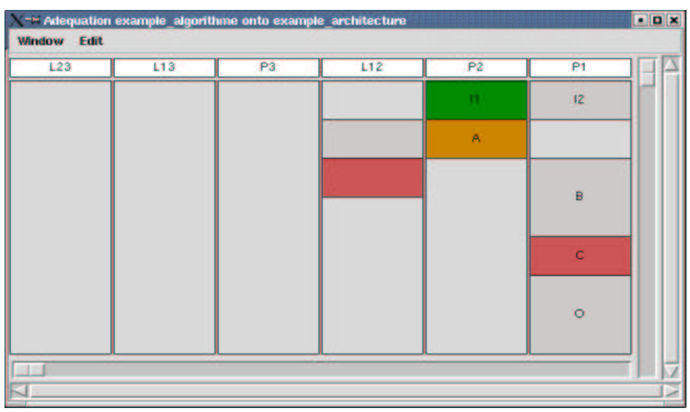


FIG. 9.5 – Distribution/ordonnancement générée par l'heuristique de AAA.

9.3.2 Heuristique tolérante aux fautes de la méthodologie AAA-TP

L'application de l'heuristique de AAA-TP sur le graphe d'algorithme $\mathcal{A}lg$ de la figure 9.2 et le graphe d'architecture $\mathcal{A}rc$ de la figure 9.3 génère la distribution/ordonnancement de la figure 9.6. Cette distribution/ordonnancement tolère une seule faute arbitraire d'un processeur ($\mathcal{N}pf = 1$ et

$\mathcal{N}lf = 0$). La longueur $\mathcal{R}tc_{cal}$ de cette distribution/ordonnancement est égale à 12. Dans cette figure, l'opération o_k# désigne la k^{ieme} réplique de l'opération o_k.

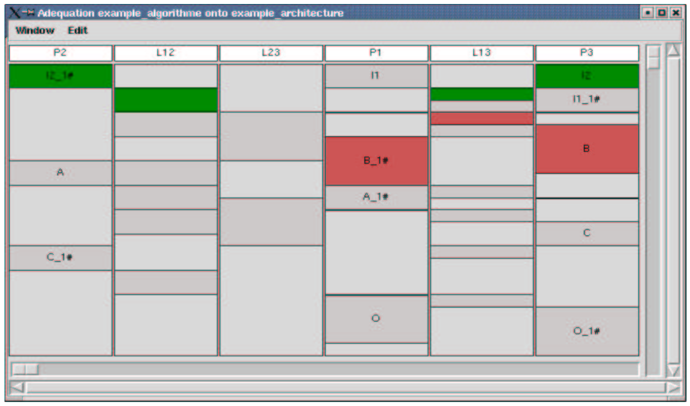


FIG. 9.6 – Distribution/ordonnancement générée par l'heuristique de AAA-TP.

9.3.3 Heuristique tolérante aux fautes de la méthodologie AAA-TB

L'application de l'heuristique de AAA-TB sur le graphe d'algorithme $\mathcal{A}lg$ de la figure 9.2 et le graphe d'architecture $\mathcal{A}rc$ de la figure 9.3 (en remplaçant chaque lien point-à-point par un bus de communication) génère la distribution/ordonnancement de la figure 9.7. Cette distribution/ordonnancement tolère une seule faute arbitraire d'un processeur ($\mathcal{N}pf = 1$ et $\mathcal{N}bf = 0$). La longueur $\mathcal{R}tc_{cal}$ de cette distribution/ordonnancement est égale à 7.5.

FIG. 9.7 – Distribution/ordonnancement générée par l'heuristique de AAA-TB.

9.3.4 Heuristique bi-critères de la méthodologie AAA-F

L'application de l'heuristique de AAA-F sur le graphe d'algorithme $\mathcal{A}lg$ de la figure 9.2 et le graphe d'architecture $\mathcal{A}rc$ de la figure 9.3 génère la distribution/ordonnancement de la figure 9.8. Les deux critères à satisfaire par cette heuristique bi-critères sont la longueur ($\mathcal{R}tc_{obj}$=7) et la fiabilité ($\mathcal{F}_{obj}$=0.999947) de la distribution/ordonnancement générée par AAA. Les taux de défaillance des processeurs P1, P2 et P3 sont respectivement $\lambda_1 = 10^{-5}$, $\lambda_2 = 10^{-6}$, $\lambda_3 = 10^{-7}$. Ainsi, les taux de défaillance des liens de communication L12, L13 et L23 sont respectivement $\mu_{12} = 10^{-5}$, $\mu_{13} = 10^{-6}$, $\mu_{23} = 10^{-7}$. Surtout, le taux maximal de réplication a été fixé à 1 pour établir une comparaison avec l'heuristique de AAA.

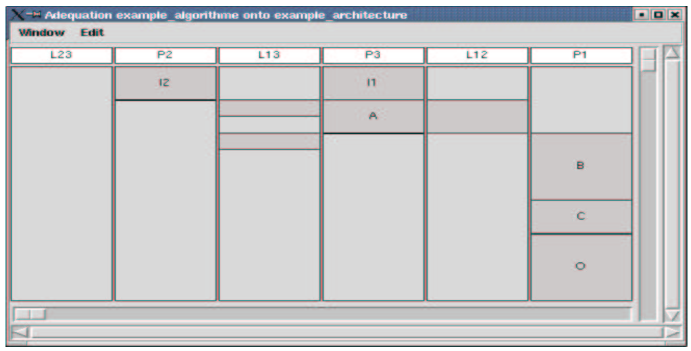


FIG. 9.8 – Distribution/ordonnancement générée par l'heuristique de AAA-F.

La longueur $\mathcal{R}tc_{cal}$ (resp. fiabilité $\mathcal{F}_{cal}$) de cette distribution/ordonnancement est égale à 7 (resp. à 0.999948). Notre heuristique trouve donc la même longueur que l'heuristique de AAA, mais avec une meilleure fiabilité.

9.4 Génération automatique d'exécutifs

Lorsque toutes les contraintes matérielles et temps réel sont satisfaites, alors SYNDEX génère automatiquement un macro-exécutif pour chaque processeur du graphe d'architecture $\mathcal{A}rc$. Ce macro-exécutif est un fichier « .m4 » [72] décrivant les séquences de calcul et de communication associées à chaque processeur. Par exemple, la figure 9.9 présente la capture d'écran de SYNDEX, cette figure correspond au macro exécutif du processeur P3.

Enfin, afin de générer un code compilable, le macro-processeur GNU M4 est utilisé pour transformer tous les fichiers « .m4 » en un code compilable en C. La figure 9.10 montre les principales phases de la génération d'un tel exécutif par SYNDEX.

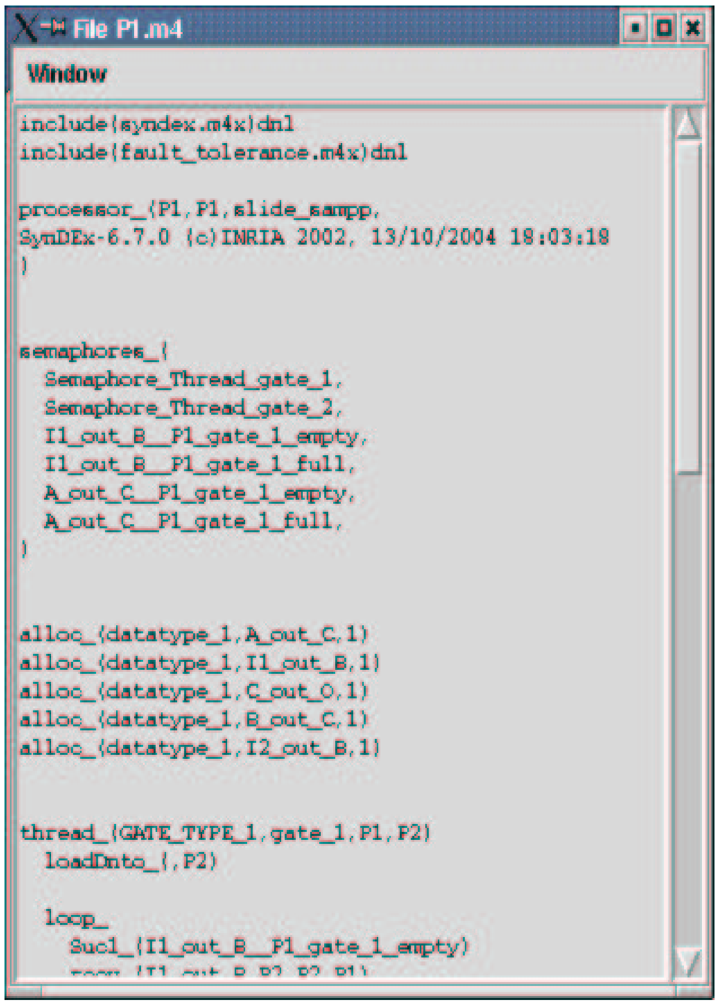


FIG. 9.9 – Exemple d'un macro-exécutif du processeur P1.

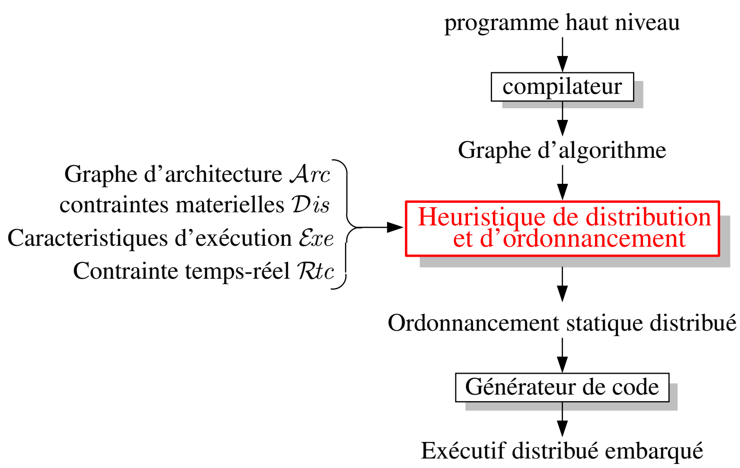


FIG. 9.10 – Processus de génération d'un exécutif par SYNDEX.

« Le hasard, dans certains cas,
c'est la volonté des autres. »
CAPUS

Chapitre 10

Génération aléatoire de graphes d'algorithmes et d'architectures

Résumé

Ce chapitre présente le générateur de graphes aléatoires qui nous a permis d'évaluer les heuristiques de distribution/ordonnancement de AAA-TP, AAA-TB et AAA-F.

Afin de mener une étude comparative des heuristiques de distribution/ordonnancement que nous avons proposées dans ce travail, il a été nécessaire de disposer d'un ensemble de graphes d'algorithmes et de graphes d'architectures. Pour cela, nous présentons dans cette section, deux générateurs aléatoires de graphes qui nous ont permis d'obtenir ces graphes. Le premier générateur génère des graphes d'algorithmes, et le deuxième génère des graphes d'architectures. Puisque nous visons des architectures hétérogènes et non forcément complètement connectées, la durée d'exécution d'un composant logiciel peut être différente d'un composant matériel à un autre, d'où la nécessité d'avoir une connexion entre ces deux générateurs. Ces deux générateurs représentent 2000 lignes de code OCaml. Le processus de génération des graphes aléatoires d'algorithmes et d'architectures est illustré sur la figure 10.1.

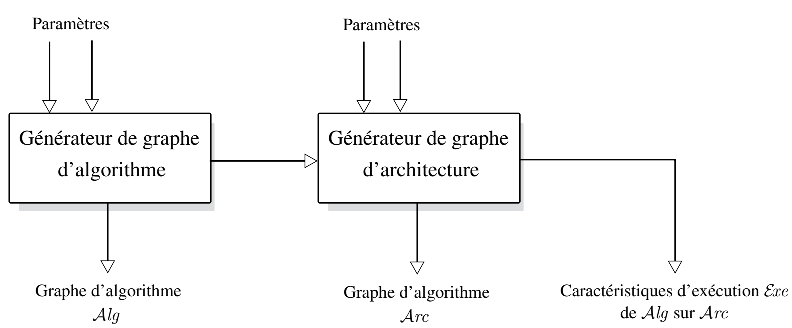


FIG. 10.1 – Processus de génération aléatoire de graphes.

10.1 Générateur de graphe d'algorithme

Le générateur que nous proposons s'inspire des travaux précédents de Hutton et al. [57]. Nous utilisons une technique de génération par niveaux, c'est-à-dire que le graphe d'algorithme est représenté par plusieurs niveaux (≥ 2). Chaque niveau i est composé de plusieurs nœuds (opérations), et chaque nœud o_i de niveau i possède au moins un prédécesseur o_j de niveau inférieur j avec $i > j$.

Ce générateur est basé sur trois paramètres (voir figure 10.2a) :

1. la taille du graphe n : c'est le nombre de nœuds dans le graphe ;
2. la hauteur maximale du graphe h : c'est le nombre maximum de niveaux dans le graphe ;
3. la largeur maximale du graphe l : c'est le nombre maximum de nœuds indépendants dans un niveau du graphe.

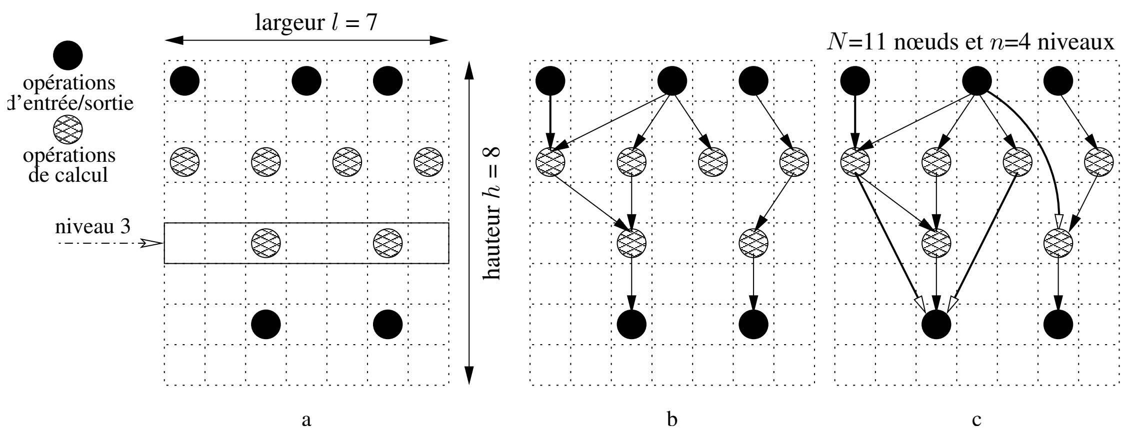


FIG. 10.2 – Étapes de génération aléatoire d'un graphe d'algorithme.

Le processus de génération d'un tel graphe d'algorithme $\mathcal{A}lg$ est réalisé en deux phases complémentaires : une phase de génération de nœuds (ou opérations) et une phase de génération d'arcs (ou dépendances de données).

Génération de nœuds : Elle est réalisée en deux étapes (voir figure 10.2a) :

- nous tirons aléatoirement n nœuds dans une matrice de dimension $[l * h]$; l et h agissent sur la forme du graphe,
- ensuite, nous construisons les N niveaux du graphe ; chaque niveau k est composé des nœuds situés sur une même ligne horizontale dans la matrice.

Génération d'arcs : Nous avons choisi de générer deux types d'arcs (définition 24) : avec et sans diffusion. La génération est réalisée en deux étapes :

- nous choisissons aléatoirement pour chaque nœud o_i de niveau i un ou plusieurs nœuds comme prédécesseurs de niveau j, avec $j = i - 1$ (voir figure 10.2b),
- ensuite, nous choisissons aléatoirement pour chaque nœud o_i de niveau i un ou plusieurs nœuds comme prédécesseurs de niveau j, avec $j < i - 1$ (voir figure 10.2c).

Le nombre de prédécesseurs, d'arcs, de niveaux et de nœuds par chaque niveau peut être contrôlé dans notre générateur par plusieurs paramètres. Par exemple, le graphe de la figure 10.2a est un graphe algorithme $\mathcal{A}lg$ composé de 11 nœuds distribués aléatoirement sur 8 niveaux au maximum, avec 7 nœuds au maximum dans chaque niveau, et au maximum 3 prédécesseurs pour chaque nœud. Tous les arcs sont de type dépendance de données sans diffusion.

10.2 Générateur de graphe d'architecture

Nous avons choisi d'utiliser la méthode de Waxman [78] pour la génération aléatoire de graphes d'architectures avec des liaisons point-à-point. Le principe de la méthode est de :

- tirer aléatoirement v nœuds de type processeurs dans une matrice,
- calculer la probabilité $P(x, y)$ d'ajouter un lien entre chaque paire de processeurs x et y :

$$P(x, y) = \beta\, e^{-\frac{d(x,y)}{\alpha L}}$$

  où L est la distance maximale entre tous les processeurs, $\beta = 2.2$ et $\alpha = 0.15$ sont des constantes, et $d(x, y)$ est la distance euclidienne entre x et y,
- pour chaque $P(x, y)$, tirer un nombre T aléatoire entre 0 et 1. Si $T < P(x, y)$, alors ajouter un nœud m de type mémoire SAM et relier m avec x et y.

10.3 Caractérisation de graphe d'algorithme et d'architecture

La caractérisation des deux graphes d'algorithme et d'architecture consiste à associer :

- pour chaque processeur les opérations qu'il peut exécuter avec leur coût d'exécution ;
- pour chaque dépendance de données la taille des données à transférer ;
- pour chaque média de communication le temps nécessaire pour transférer les données de chaque dépendance de données.

Le processus de caractérisation de ces graphes dépend de trois paramètres, qui permettent d'obtenir différentes caractéristiques d'exécution $\mathcal{E}xe$ d'un graphe d'algorithme $\mathcal{A}lg$ sur un graphe d'architecture $\mathcal{A}rc$. Ces paramètres sont :

1. les coûts d'exécution « CE » : ils définissent deux bornes pour les coûts d'exécution de chaque opération sur chaque processeur : coût d'exécution maximal CE_{max} et minimal CE_{min} ;

2. les coûts de communication « CC » : ils définissent deux bornes pour les coûts de communication des données sur chaque média de communication : coût de communication maximal CC_{max} et minimal CC_{min} ;
3. le ratio entre les coûts de communication et les coûts d'exécution « CCR » : le choix de la valeur de ce paramètre a des conséquences importantes sur l'efficacité d'une heuristique de distribution et d'ordonnancement. Il est défini par la formule suivante :

$$CCR = \frac{CC_{max}}{CE_{max}}$$

Un CCR supérieur à 1 indique que les communications coûtent « plus cher » que les calculs. Au cours de l'évaluation des algorithmes de distribution/ordonnancement, il est intéressant de tester *plusieurs* valeurs de CCR.

« La vérité de demain se nourrit de l'erreur d'hier. »
Antoine de SAINT-EXUPÉRY

Conclusion et perspectives

Les travaux présentés dans cette thèse s'inscrivent dans l'objectif global de la conception de systèmes réactifs sûrs de fonctionnement. Plus particulièrement, les objectifs de ce travail étaient de résoudre deux problèmes de distribution et d'ordonnancement temps réel des composants logiciels d'un algorithme sur les composants matériels d'une architecture. Le premier problème consistait à chercher une distribution/ordonnancement tolérante aux fautes des processeurs et des média de communication. Le deuxième problème consistait à rendre une distribution/ordonnancement la plus fiable possible.

Le problème de la distribution/ordonnancement temps réel et tolérant aux fautes (resp. fiable) est un problème d'optimisation NP-difficile, puisque il s'agit ici de trouver une solution qui minimise le temps global de l'exécution d'un algorithme sur une architecture distribuée et hétérogène, tout en respectant des contraintes matérielles de temps réel, et en tolérant des fautes matérielles (resp. en maximisant la fiabilité de la distribution/ordonnancement). La solution optimale à ce problème NP-difficile ne peut être trouvée que par des algorithmes exacts de complexité exponentielle ; c'est pourquoi nous avons proposé d'utiliser dans ce travail des heuristiques qui cherchent une solution si possible proche de la solution optimale, tout en étant de complexité polynômiale.

Puisque nous visons des systèmes embarqués et pour des raisons de coûts de conception, les solutions que nous avons proposées dans ce travail sont basées sur des techniques logicielles, c'est-à-dire qu'elles utilisent au mieux la redondance matérielle existante au niveau de l'architecture du système sans essayer de rajouter des ressources physiques supplémentaires. Par conséquent, la redondance logicielle des composants logiciels de l'algorithme a été choisie comme technique de redondance, ce qui nous a semblé le plus approprié pour pouvoir atteindre hors-ligne les deux objectifs de tolérance aux fautes et de fiabilité.

Les résultats des travaux de recherche effectués pendant trois ans de thèse ont donné lieu à trois méthodologies de conception de systèmes distribués réactif, embarqués, fiables et tolérants aux fautes. Elles sont basées sur des heuristiques de distribution/ordonnancement temps réel et sur le principe de la redondance logicielle. Les deux premières méthodologies proposent deux solutions au problème de la tolérance aux fautes, la première solution étant adaptée aux architectures à liaisons point-à-point, tandis que la deuxième solution est adaptée aux architectures à liaisons bus. La troisième méthodologie propose une solution au problème de la fiabilité d'un système.

La première méthodologie, que nous avons appelée AAA-TP, génère automatiquement des distributions/ordonnancements tolérantes à n'importe quelle combinaison de plusieurs fautes matérielles. Les fautes que nous avons considérées sont des fautes temporelles des processeurs et des liens point-à-point de communication. La tolérance aux fautes est obtenue hors-ligne par l'utilisation de la redondance active des composants logiciels de l'algorithme.

La deuxième méthodologie, que nous avons appelée AAA-TB, génère automatiquement des distributions/ordonnancements tolérantes à n'importe quelle combinaison de plusieurs fautes matérielles. Les fautes que nous avons considérées sont des fautes temporelles des processeurs et des bus de communication. Cette fois, la tolérance aux fautes est obtenue hors-ligne par l'utilisation de la redondance hybride des composants logiciels de l'algorithme et par la fragmentation des données de communication en plusieurs paquets de données.

La troisième méthodologie, que nous avons appelée AAA-F, génère automatiquement des distributions/ordonnancements fiables. Elle est basée sur la redondance active des composants logiciels de l'algorithme et sur une heuristique de distribution/ordonnancement bi-critères. La redondance logicielle est utilisée dans le but de maximiser la fiabilité d'une distribution/ordonnancement. L'heuristique bi-critères consiste à optimiser deux objectifs antagonistes, qui sont la minimisation de la longueur et la maximisation de la fiabilité d'une telle distribution/ordonnancement.

Les distributions/ordonnancements temps réel, tolérantes aux fautes et fiables générées par les trois méthodologies AAA-TP, AAA-TB et AAA-F sont toutes prédictives, c'est-à-dire que les contraintes de temps réel et de fiabilité peuvent être vérifiées avant la mise en exploitation du système, à la fois en l'absence et en présence de défaillances des processeurs et des média de communication.

Enfin, les travaux présentés dans cette thèse offrent un certain nombre de perspectives :

- Avant tout, les trois heuristiques de AAA-TP, AAA-TB et AAA-F, doivent être testées plus extensivement sur des graphes d'algorithmes et d'architectures générés aléatoirement afin de mieux comprendre l'influence des paramètres. De plus, la fragmentation des données dans la méthodologie AAA-TB pourrait être améliorée pour prendre en compte l'ensemble de tous les bus disponibles dans l'architecture.
- Ensuite, ces méthodologies doivent être testées sur des cas réalistes. Par exemple, la méthodologie AAA-TB peut être utilisée dans le cas des Cycab, petits véhicules urbains développés par l'INRIA [47]. Ce véhicule électrique dispose de quatre roues motrices et directrices pilotées par un système informatique embarqué. Son architecture matérielle est composée de six processeurs et d'un unique bus de communication CAN. Jusqu'à présent nous avons implanté nos trois méthodologies dans le logiciel SynDEx, donc il reste à adapter le générateur automatique d'exécutifs de SynDEx à la méthodologie AAA-TB, afin de remplacer le système actuel des Cycab par un système embarqué tolérant aux fautes des processeurs. Pour tolérer en plus des fautes des processeurs des fautes de bus de communication, l'architecture matérielle des Cycab devra être modifiée par l'ajout de bus supplémentaires. Le nombre de bus à ajouter dépendra du nombre de fautes de bus à tolérer.

- Les deux méthodologies AAA-TP et AAA-TB peuvent être enrichies par des nouveaux moyens de sûreté de fonctionnement, tels que le *fonctionnement en mode dégradé*. Dans le cas où les hypothèses de défaillance d'un système ne sont pas respectées, par exemple en présence de trop de défaillances de processeurs, le système peut continuer à fonctionner mais en mode dégradé, c'est-à-dire qu'il n'assure que certaines de ses fonctionalitées. Actuellement, la plupart des nouveaux systèmes industriels, tolérants ou non aux fautes, sont équipés d'un mécanisme de basculement du fonctionnement en mode normal vers plusieurs fonctionnements en mode dégradé. Les questions qui se posent dans cette conception sont : comment spécifier les modes dégradés ? Comment prendre en compte le mode normal et le mode dégradé en même temps dans nos heuristiques de distribution/ordonnancement ? Comment gérer le passage du mode normal à un mode dégradé ? ...
- Dans certains systèmes, la présence de fautes des liens de communication point-à-point peut diviser l'architecture matérielle du système en plusieurs sous-architectures *disjointes*. Il serait intéressant de prendre en compte ce problème des sous-architectures disjointes dans la méthodologie AAA-TP.
- Enfin, un élément essentiel des systèmes réactifs embarqués sont les capteurs sensibles aux valeurs issues de l'environnement extérieur contrôlé. Des exemples de tels systèmes sont la télésurveillance de la santé et de la sécurité des personnes malades. Au vu des conséquences catastrophiques que pourrait entraîner une faute d'un capteur, la conception d'architecture multi-capteurs est vitale pour ce type de systèmes. Les questions qui se posent dans cette conception sont : quel devrait être le type de la redondance physique des capteurs (capteurs avec même modalités, avec modalités différentes ou capteurs complémentaires) ? Quel type d'algorithme permettrait d'obtenir les compromis entre les valeurs issues des capteurs redondants (voteur ou fusion de données) ? ... Et surtout comment adapter nos heuristiques de distribution/ordonnancement à ce problème spécifique des fautes des capteurs.

Bibliographie

[1] A. Abd-allah. Extending reliability block diagrams to software architectures. Technical report, Center for Software Engineering, Computer Science Department, University of Southern California, Los Angeles, CA 90089 USA, 1997.

[2] I. Ahmad and Y.K. Kwok. On exploiting task duplication in parallel program scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, volume 9, pages 872–892, September 1998.

[3] I. Ahmad, Y.K. Kwok, and M.Y. Wu. Performance comparison of algorithms for static scheduling of dags to multiprocessors. In *Proceedings of the 2nd Australian Conference on Parallel and Real-Time Systems*, pages 185–192, Sep 1995.

[4] `http://www.cnes.fr/espace_pro/communiques/cp96/rapport_501/rapport_501_2.html`.

[5] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristics for distributed embedded systems under reliability and real-time constraints. In *International Conference on Dependable Systems and Networks, DSN'04*, Firenze, Italy, June 2004. IEEE.

[6] N. Auluck and D. P. Agrawal. Reliability driven, non-preemptive real time scheduling of periodic tasks on heterogeneous systems. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages pp. 803–809, MIT, Cambridge, USA, November 4-6 2002.

[7] A. Avizienis. Design of fault-tolerant computers. In *Fall Joint Computer*, pages 733–743, 1967.

[8] A. Avizienis. Dependable systems of the future : What is still needed ? In *Building the information society, 18th IFIP World Computer Congress*, pages 79–90, Toulouse, France, August 2004. Kluwer Academic Publishers.

[9] A. Avizienis, J.-C. Laprie, and B. Randell. Dependability and its threats : a taxonomy. In *Building the information society, 18th IFIP World Computer Congress*, pages 91–120, Toulouse, France, August 2004. Kluwer Academic Publishers.

[10] A. Banerjea. Simulation study of the capacity effects of dispersity routing for fault-tolerant real-time channels. In *SIGCOMM Symposium*, pages 194–205, August 1996.

[11] J.-P. Beauvais. *Étude d'algorithmes de placement de tâches temps-réel périodiques complexes dans un système réparti*. PhD thesis, École Centrale de Nantes, June 1996.

[12] G. Bracha. Resilient consensus protocols. *ACM*, 1983.

[13] A. Burns and A. Wellings. *Real-time systems and programming languages*. Addison-Wesley, 1997.

[14] E. Cabau. Introdution à la conception de la sûreté. Technical report, Schneider Electric, 1999.

[15] B. Chen, S. Kamat, and W. Zhao. Fault-tolerant real-time communication in FDDI-based networks. In *IEEE Real-Time Systems Symposium*, pages 141–151, 1995.

[16] D.J. Chen, M.S Chang, M.C. Sheng, and M.S. Horng. Time constrained distributed program reliability analysis. *Journal of Information Sciences and Engineering*, VOL. 117(NO. 14 pp. 891-911), 1998.

[17] P. Chevochot and I. Puaut. Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategie. In *The 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 356–363, HongKong, China, December 1999.

[18] A. Colin, I. Puaut, C. Rochange, and P. Sainrat. Calcul de majorants de pire temps d'exécution : état de l'art. *Techniques et Sciences Informatiques (TSI)*, 22(5) :651–677, 2003.

[19] C. Constantinescu. Impact of deep submicro technology on dependability of VLSI circuits. In *International Conference on Dependable Systems and Networks, DSN'02*, 2002.

[20] M. Cosnard and A. Ferreira. On the real power of loosely coupled parallel architectures. *Parallel Processing Letters*, 1(2) :103–112, 1991.

[21] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel. Off-line real-time fault-tolerant scheduling. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 410–417, Mantova, Italy, February 2001.

[22] A. Dogan and F. Özgüner. Optimal and suboptimal reliable scheduling of precedence-constrained tasks in heterogeneous distributed computing. In *Proceedings of the 2000 International Conference on Parallel Processeing (ICPP00-Workshops)*, Toronto, Canada, August 2000.

[23] S. Dulman, T. Nieberg, J. Wu, and P. Havinga. Trade-off between traffic overhead and reliability in multipath routing for wireless sensor networks. In *Wireless Communications and Networking Conference*, 2003.

[24] G. Durairaj. *Evaluating the reliability of Distributed Real-Time Systems*. PhD thesis, University of Massachusetts, 1999.

[25] P. Felber, X. Défago, P. Eugster, and A. Schiper. Replicating CORBA objects : a marriage between active and passive replication. In *Second IFIP International Working Conference on Distributed Applications and Interoperable Systems (DAIS'99)*, pages 375–387, Helsinki, Finland, 1999.

[26] P. Fragopoulou and S. G. Akl. Fault-tolerant communication algorithms on the star network using disjoint paths. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, Kingston, Ontario, Canada, 1995.

[27] C. Gagné, M. Gravel, and W. L. Price. Recherche de solutions de compromis en ordonnancement à l'aide de metaheuristiques. In *Quatrième conférence francophone de modélisation et simulation - MOSIM'03*, Toulouse, France, April 2003.

[28] M.R. Garey and D.S. Johnson. *Computers and intractability, a guide to the theory of NP-completeness*. W.H. Freeman Company, San Francisco, 1979.

[29] A. Girault, H. Kalla, M. Sighireanu, and Y. Sorel. An algorithm for automatically obtaining distributed and fault-tolerant staassatic schedule. In *The International Conference on Dependable Systems and Networks*, San Francisco, California, USA, June 2003.

[30] A. Girault, H. Kalla, and Y. Sorel. Une heuristique d'ordonnancement et de distribution tolérante aux pannes pour systèmes temps-réel embarqués. In *Modélisation des Systèmes Réactifs, MSR'03*, pages 145–160, Metz, France, October 2003. Hermes.

[31] A. Girault, H. Kalla, and Y. Sorel. An active replication scheme that tolerates failures in distributed embedded real-time systems. In *IFIP Working Conference on Distributed and Parallel Embedded Systems, DIPES'04*, Toulouse, France, August 2004. Kluwer Academic.

[32] A. Girault, H. Kalla, and Y. Sorel. A scheduling heuristic for distributed real-time embedded systems tolerant to processor and communication media failures. *International Journal of Production Research*, 42(14) :2877–2898, July 2004.

[33] A. Girault, C. Lavarenne, M. Sighireanu, and Y. Sorel. Fault-tolerant static scheduling for real-time distributed embedded systems. In *21st International Conference on Distributed Computing Systems, ICDCS'01*, pages 695–698, Phœnix, USA, April 2001. IEEE. Extended abstract.

[34] T. Grandpierre. *Modélisation d´architecture parallèles hétérogènes pour la génération automatique d´exécutifs distribués temps réel optimisés*. PhD thesis, Université Paris XI, ORSAY, 2000.

[35] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessor. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, may 1999.

[36] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Proceeding Conference on Reliable Software Technologies*, pages 38–57. Springer-Verlag, 1996.

[37] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *Computer*, 30(4) :68–74, April 1997.

[38] S. Han and K.G. Shin. Fast restoration of real-time communication service from component failures in multi-hop networks. In *SIGCOMM Symposium*, September 1997.

[39] K. Hashimoto, T. Tsuchiya, and T. Kikuno. Effective scheduling of duplicated tasks for fault-tolerance in multiprocessor systems. *IEICE Transactions on Information and Systems*, E85-D(3) :525–534, March 2002.

[40] Y. He, Z. Shao, B. Xiao, Q. Zhuge, and E. Sha. Reliability driven task scheduling for tightly coupled heterogeneous systems. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina Del Ray, USA, November 2003.

[41] M. Hiller. Software fault-tolerance techniques from a real-time systems point of view. Technical report, Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Göteborg Sweden, November 1998.

[42] P. Jalote. *Fault-Tolerance in Distributed Systems*. Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[43] N. Kandasamy, J. P. Hayes, and B.T. Murray. Dependable communication synthesis for distributed embedded systems. In *22nd Int'l Conf. on Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, UK, September 2003.

[44] B. Kao, H. Garcia-Molina, and D. Barbara. Aggressive transmissions of short messages over redundant paths. In *IEEE Transactions on Parallel and Distributed Systems*, volume 5, pages 102–109, January 1994.

[45] S. Kartik and C. Siva Ram Murthy. Improved task allocation algorithms to maximize reliability of redundant distributed computing systems. *IEEE Transactions On Reliability*, 44(4), December 1995.

[46] S. Kartik and C. Siva Ram Murthy. Task allocation algorithms for maximizing reliability of distributed computing systems. *IEEE Transactions On Computers*, 41(9), September 1997.

[47] R. Kocik. *Sur l´optimisation des systèmes distribués temps-réel embarqués : application au protypage rapide d´un véhicule électrique autonome*. PhD thesis, Université de Rouen U.F.R de Sciences, March 2000.

[48] H. Kopetz. The fault hypothesis for the time-triggered architecture. In *Building the information society, 18th IFIP World Computer Congress*, pages 220–233, Toulouse, France, August 2004. Kluwer Academic Publishers.

[49] P.B. Lakey and A.M. Neufelder. *System and Software Reliability Assurance Notebook*. Softrel, 1997.

[50] J. C. Laprie, editor. *Dependability : Basic Concepts and Terminology*. Springer-Verlag Wien New York, 1991.

[51] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The syndex software environment for real-time distributed systems design and implementation. In *European Control Conference, ECC'9*, July 1991.

[52] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement de signal*, 14(6), 1997.

[53] J.K. Lenstra and A.H.G. Rinnoy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1) :22–35, 1978.

[54] M.S. Lin, M.S Chang, and D.J. Chen. Efficient algorithms for reliability analysis of distributed computing systems. *Information Sciences*, 117(1-2) :89–106, 1999.

[55] M.S. Lin, D.J. Chen, and M.S. Horng. The reliability analysis of distributed computing systems with imperfect nodes. *The Computer Journal*, VOL. 42, 1999.

[56] O. Lundkvist. *Implementation of Fault-Tolerance Techniques for Real-Time Multiprocessor Scheduling*. PhD thesis, Departement of Computer Engineering, Chalmers University of Technoloy, Göteborg, December 1997.

[57] J. Rose M. D. Hutton, J. P. Grossman and D. G. Corneil. Characterization and parameterized random generation of digital circuits. In *Design Automation Conference*, pages 94–99, 1996.

[58] K. Marzullo and M. Wood. Making real-time reactive systems reliable. In *Fourth European SIGOPS Workshop*, 1990.

[59] J. K. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction.

[60] L'usine nouvelle. Spécial industrie 2004 : mécanique, toujours plus de performances. magasine, `http://www.usinenouvelle.com`, March 2004.

[61] Y. Oh and S. H. Son. Scheduling real-time tasks for dependability. *Journal of Operational Research Society*, 48(6) :629–639, June 1997.

[62] B. Parhami. Voting algorithms. *IEEE transactions on reliability*, 43(4) :617–629, 1994.

[63] D. Powell. Failure mode assumptions and assumption coverage. In *International Symposium on Fault-Tolerant Computing*, July 1992.

[64] X. Qin, Z.F. Han, H. Jin, L. P. Pang, and S. L. Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *Proceeding of the International Workshop on Cluster Computing-Technologies, Environments, and Applications (CC-TEA'2000)*, Las Vegas, USA, June 2000.

[65] X. Qin and H. Jiang. Dynamic, reliability-driven scheduling of parallel real-time jobs in heterogeneous systems. In *Proceedings of the 30th International Conference on Parallel Processing (ICPP 2001*, pages 113–122, Valencia, Spain, September 2001.

[66] X. Qin, H. Jiang, and D. R. Swanson. An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems. In *Proceedings of the 31th International Conference on Parallel Processing (ICPP 2002)*, pages 360–386, Vancouver, British Columbia, Canada, August 2002.

[67] P. Ramanathan and K.G. Shin. Delivery of time-critical messages using a multiple copy approach. *ACM Transactions on Computer Systems*, 10(2) :144–166, May 1992.

[68] J. Reisinger and A. Steininger. The design of a fail-silent processing node for the predictable hard real-time system MARS, 1993.

[69] J. Rushby. Critical system properties : Survey and taxonomy. *Reliability Engineering and System Safet*, 43(2) :189–219, 1994.

[70] R.A. Sahner and K.S. Trivedi. A hierarchial, combinatorial-markov model of solving complex reliability models. In *Proceedings of 1986 ACM Fall joint computer conference*, pages 817–825, 1986.

[71] R.D. Schilichting and F.B. Schneider. Fail stop processors : An approach to designing fault tolerant computing systems. *ACM Transactions on Computer Systems*, 3(1) :145–154, August 1983.

[72] R. Seindal. Gnu m4, a power ful macro processor. `http://www.cs.wisc.edu/csl/texihtml/m4-1.4/m_4toc.html`.

[73] S.M. Shatz, J.P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. In *IEEE Trans. Computers*, volume 41, pages 156–168, September 1992.

[74] Y. Sorel. Massively parallel computing systems with real time constraints - the algorithm architecture adequation methodology. In *Massively Parallel Computing Systems*, may 1994.

[75] Y. Sorel. Syndex : un logiciel de cao niveau système pour l aide à l implantation d applications distribuées temps réel embarquées. In *quatrièmes Journées Nationales de la Recherche en Robotique (JNRR 03)*, 2003.

[76] S. Srinivasan and N.K. Jha. Safety and reliability driven task allocation in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(3) :238–251, March 1999.

[77] N. Suri and K. Ramamritham. Editorial : Special section on dependable real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :529–531, June 1999.

[78] M. Thomas and E. W. Zegur. Generation and analysis of random graphs to model internetworks. Technical report, College of Computing Georgia Institute of Technology, 1994.

[79] W. Torres-Pomales. Software fault tolerance : A tutorial, October 2000. National Aeronautics and Space Administration (NASA) Langley Research Center.

[80] P. Traverse, I. Lacaze, and J. Souyris. Airbus fly-by-wire : a total approach to dependability. In *Building the information society, 18th IFIP World Computer Congress*, pages 191–212, Toulouse, France, August 2004. Kluwer Academic Publishers.

[81] A. Vicard. *Formalisation et optimisation des systèmes informatiques distribués temps-réel embarqués*. PhD thesis, Université Paris XIII, July 1999.

[82] R. A. Walker and S. Chaudhuri. High-level synthesis : Introduction to the scheduling problem. In *IEEE Design & Test*, 1995.

[83] T. Yang and A. Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12) :1321–1344, 1993.

[84] Y. C. Yeh. Unique dependability issues for commercial airplane fly-by-wire systems. In *Building the information society, 18th IFIP World Computer Congress*, pages 213–220, Toulouse, France, August 2004. Kluwer Academic Publishers.

[85] Q. Zheng and K. G. Shin. Fault-tolerant real-time communication in distributed computing systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 470–480, 1998.

« Génération automatique de distributions/ordonnancements temps réel, fiables et tolérants aux fautes »

Résumé : Les systèmes réactifs sont de plus en plus présents dans de nombreux secteurs d´activité tels que l´automobile, les télécommunications et l´aéronautique. Ces systèmes réalisent des tâches complexes qui sont souvent critiques. Au vu des conséquences catastrophiques que pourrait entraîner une défaillance dans ces systèmes, suite à la présence de fautes matérielles (processeurs et média de communication), il est essentiel de prendre en compte la tolérance aux fautes dans leur conception. En outre, plusieurs domaines exigent une évaluation quantitative du comportement de ces systèmes par rapport à l'occurrence et à l'activation des fautes. Afin de concevoir des systèmes sûrs de fonctionnement, j'ai proposé dans cette thèse trois méthodologies de conception basées sur la théorie d'ordonnancement et la redondance active et passive des composants logiciels du système. Ces trois méthodologies permettent de résoudre le problème de la génération automatique de distribution et d'ordonnancements temps réel, fiables et tolérants aux fautes. Ce problème étant NP-difficile, ces trois méthodologies sont basées sur des heuristiques de type ordonnancement de liste. Plus particulièrement, les deux premières méthodologies traitent le problème de la tolérance aux fautes matérielles des processeurs et des media de communication, respectivement pour des architectures à liaisons point-à-point et des architectures à liaison bus. La troisième méthodologie traite le problème de l'évaluation quantitative d'une distribution/ordonnancement en terme de fiabilité à l'aide d'une heuristique bi-critère originale. Ces méthodologies offrent de bonnes performances sur des graphes d'algorithme et d'architecture générés aléatoirement.

Mots-clés : systèmes distribués temps réel embarquées, systèmes critiques, architectures réparties hétérogènes, heuristiques de distribution/ordonnancement, tolérance aux fautes, fiabilité, fautes transitoires, redondance logicielle.

« automatic generation of distributed, real-time, fault-tolerant and reliable schedules »

Abstract : Reactive systems are increasingly used in fields such as automotive, telecommunication, and aeronautic. These systems carry out complex tasks which are often critical. Within catastrophic consequences that could involve a fault in these systems, due to the presence of hardware fault (processor and communication media), it is essential to take into account fault-tolerance in their design. Moreover, several fields require a quantitative evaluation of their system behavior with respect to fault occurrence and fault activation. In order to design dependable systems, I propose in this thesis three design methodologies, based on scheduling theory, and on active and passive software redundancy. These three methodologies allow me to solve the problem of automatic generation of fault-tolerant real-time and reliable schedules. This problem is NP-hard, so these three methodologies are based on list scheduling heuristics. More precisely, the first two methodologies deal with the problem of hardware fault-tolerance (processors and communication media faults), respectively for architectures with point-to-point and buses communication links. The third methodology deals with the problem of quantitative evaluation of schedules in terms of reliability through an original bi-criteria heuristic. These methodologies offer good performances on algorithm and architecture graphs radomly generated.

Keywords : distributed embedded real-time systems, critical systems, heterogeneous distributed architectures, distribution and scheduling heuristics, fault-tolerance, reliability, transient faults, software redundancy.